UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE GRADO

EVOLUCIÓN Y ASEGURAMIENTO DE LA CALIDAD EN UAITHNE E IMPLANTACIÓN EN UNIT LINKED

Josué Fernández León
Tutor: José Antonio Clavijo Blázquez
Ponente: Francisco Saiz López

JULIO 2014

Agradecimientos

A la empresa Delonia Software donde he realizado el trabajo de fin de grado y en especial a José Antonio y a Juan Luis por enseñarme muchísimas cosas nuevas y darme la confianza que necesitaba.

A aquellos profesores que he tenido durante estos años y que me han aportado no solo múltiples conocimientos, sino que me han demostrado que son grandes personas que se preocupan por sus alumnos.

A todos mis amigos, a los que he conocido en la universidad y me han sacado muchas sonrisas estos años y me han apoyado muchísimo y por supuesto a los que siempre han estado y están ahí tanto en los buenos momentos como en los malos.

Y muy especialmente a mi familia por confiar en todo lo que hago y no dejar que me venga abajo en los momentos difíciles, por valorarme y demostrármelo día a día, porque sé que sin ellos habría sido mucho más difícil.

Resumen

Las empresas de desarrollo de software tratan de mantener un equilibrio entre la calidad y el coste, de manera que sea posible alcanzar los objetivos marcados en un determinado proyecto. La productividad es uno de los grandes factores a tener en cuenta y se puede mejorar, entre otras formas, reduciendo aquellas tareas repetitivas que pudieran ser automatizadas.

Por ello surge Uaithne, como framework de Java capaz de definir una arquitectura de backend que facilite el crecimiento, la modificación y la separación de competencias de un sistema; y a la vez como generador de código capaz de ahorrar gran cantidad de líneas, algunas de ellas consecuencia de la arquitectura definida y muchas otras relacionada con la implementación de acceso a base de datos.

Este proyecto se marca como objetivo mejorar la calidad de Uaithne, y por ello se realiza el diseño y desarrollo de pruebas unitarias sobre su generador y se mejora su documentación creando un manual que sirva de introducción práctica al uso de la herramienta.

Otro de los objetivos es seguir evolucionando Uaithne y debido a esta necesidad se realizan satisfactoriamente tres importantes funcionalidades como son la herencia entre entidades (o representaciones de datos), el soporte de procedimientos almacenados por parte de la herramienta, y la implementación de un mecanismo de validación de datos en las entidades generadas.

Por último se muestra una aplicación práctica de Uaithne en un proyecto real denominado Unit Linked llevado a cabo por la empresa Delonia Software, lugar donde se ha realizado el proyecto de fin de grado.

Palabras clave: Uaithne, generador de código, framework, programación orientada a aspectos, manual de uso, productividad, aseguramiento de calidad.

Abstract

Software development companies try to keep a balance between quality and price, so that would be possible to achieve the goals of a certain project. Productivity is one of the most important factors to keep in mind, and it can be improved by reducing repetitive tasks which could be automated.

That is the reason why Uaithne emerged as a Java framework that is able to define a backend architecture that facilitates the growing, modification and competence separation of a system. Besides, as a code generator can save a lot of lines, some of them are the result of the defined architecture while many others are related to the database access implementation.

This project tries to improve the quality of Uaithne. For this purpose, the design and development of unit tests have been carried out on the generator, and its documentation has been improved by creating a guide which will serve as a practical introduction to the use of the tool.

Another goal is to continue evolving Uaithne and because of this three important functionalities have been successfully performed. These are inheritance between entities (or data representations), stored procedures generation and the implementation of a data validation mechanism for generated entities.

Finally it is possible to see a practical application of Uaithne in a real project called Unit Linked conducted by the company Delonia Software, where the final project was done.

Keywords: Uaithne, code generator, framework, aspect-oriented programming, user guide, productivity, quality assurance.

Índice de contenidos

Contenido

Índice de figuras

Introducción

Motivación y objetivos

Durante las labores de implementación de aplicaciones informáticas siempre aparecen una serie de tareas repetitivas que pueden ser automatizadas con el fin de mejorar la productividad del equipo de trabajo y esto se puede conseguir mediante la utilización de herramientas de generación de código. Adicionalmente es muy importante y cada vez tiene más peso en los proyectos software el aseguramiento de la calidad de los productos implementados, ya que aumenta las posibilidades de éxito final de cualquier proyecto.

El objetivo de este proyecto es la evolución de la herramienta de generación de código Uaithne (herramienta para el desarrollo ágil del backend de aplicaciones Java) e implementación de un sistema de aseguramiento de la calidad para la misma (Pruebas unitarias, documentación, etc.); así como su incorporación en el proyecto comercial Unit Linked como aplicación práctica del mismo.

El objetivo principal se desglosa en las siguientes tareas:

- **Diseño e implantación de pruebas unitarias**, muy importantes y necesarias para poder llevar a cabo el desarrollo de un framework (en este caso Uaithne) asegurando que los cambios que se produzcan durante sus futuras evoluciones no produzcan efectos no deseados en el sistema.
- **Generación de documentación de uso de la herramienta**, que permita al lector adquirir los conocimientos necesarios para poder utilizar Uaithne en el desarrollo ágil del backend de sus proyectos Java.
- **Evolución de la funcionalidad actual**:
    - Implementación de Herencia entre "Entity Views"
    - Generación de integración con procedimientos almacenados.
    - Implementación de un mecanismo de validación de datos en las entidades generadas.

El proyecto también incluye una aplicación práctica de la solución en un proyecto comercial (Unit Linked) en desarrollo por parte de la compañía Delonia Software.

Estructura del documento

El documento se encuentra organizado de la siguiente manera:

Incluye una **primera sección** introductoria (esta misma) que comenta la motivación y los objetivos del proyecto, así como la estructura del documento.

La **segunda sección** presenta el estudio realizado sobre técnicas y tecnologías necesarias de asimilar para poder llevar a cabo la realización del proyecto.

La **tercera sección** tiene la intención de situar al lector en el contexto de Uaithne para evitar que pueda perderse durante la lectura de la siguiente sección, realizando una presentación de este sistema, definiéndolo y dejando claras cuáles son sus aportaciones.

La **cuarta sección** entra de lleno en el diseño y desarrollo del proyecto. En un primer lugar se presenta el diseño e implantación de pruebas unitarias en Uaithne y tras ello se trata sobre la generación de la documentación de uso. Seguidamente se lleva a cabo la explicación de cómo se han realizado las distintas evoluciones sobre la funcionalidad actual de Uaithne; cada evolución se divide en tres secciones: un pequeño apartado en el que se detalla la motivación de cada una de estas evoluciones, el proceso de diseño y su desarrollo. Por último se expone una aplicación práctica de Uaithne en el proyecto comercial Unit Linked.

La **quinta sección** expone las conclusiones que se han obtenido de la realización del proyecto.

Por último se encuentra un apartado con las **referencias** citadas en el documento y el **Anexo A** que se corresponde con el documento de uso de Uaithne elaborado durante la realización del proyecto.

Estudio del estado del arte

Pruebas unitarias

Son aquellas que permiten comprobar la lógica, funcionalidad y la correcta especificación de cada módulo, por separado, dentro de un sistema.
Se enfocan como pruebas de caja blanca, es decir, se centran en probar el comportamiento interno y la estructura del programa examinando la lógica interna, tratando ejecutar el mayor número de sentencias posibles, recorriendo todos los caminos independientes de cada módulo (buscando la mayor cobertura posible o la más óptima), comprobando decisiones lógicas e intentando provocar situaciones extremas. [Escuela Politécnica Superior UAM, 2013]

Para ello se crean módulos conductores y módulos resguardo. Los módulos conductores o impulsores son aquellos creados específicamente para las pruebas que se encargan de llamar a los módulos a probar. Los módulos resguardo o auxiliares son también creados específicamente para la prueba, pero son llamados por el módulo a probar.
La utilización de módulos conductores y de resguardo permiten que todos los módulos de la aplicación puedan ser probados independientemente. [Bolaños Alonso, 2008]

Junit 4.x

Es un Framework para la automatización de pruebas unitarias de programas Java [JUnit] . La versión 4.x requiere JDK 5 o superior.
Es un proyecto de código abierto escrito en Java, distribuido actualmente bajo la licencia Eclipse Public License Versión 1.0 y accesible a través del repositorio de GitHubhttps://github.com/junit-team/junit/.

Por cada clase del sistema se crea una clase de prueba con métodos (tests) destinados a probar la clase del sistema.
Cada test esta anotado con @Test y contiene el código que crea los objetos necesarios para la prueba y las aserciones que comprueban la corrección del resultado. [Escuela Politécnia Superior UAM, 2012]

El orden en el que se ejecutan los métodos de una clase de prueba [Escuela Politécnia Superior UAM, 2012] es el siguiente:

- Método con anotación @BeforeClass.
- Por cada método anotado con @Test:
    - Constructor de la clase de prueba
    - Método anotado con @Before
    - Método anotado con @Test
    - Método anotado con @After
    - Método con anotación @AfterClass

Patrones de Diseño: Patrones de Comportamiento

Los patrones de comportamiento tienen que ver con los algoritmos y las asignaciones de responsabilidad entre objetos. Estos patrones no sólo describen patrones de objetos y clases, sino que también tienen en cuenta la comunicación entre ellos.
Los patrones de comportamiento sobre clases se apoyan en la herencia para distribuir el comportamiento entre las clases, en cambio los que se aplican sobre objetos usan la composición con mayor frecuencia que la herencia. [Gamma, Helm, Johnson, & Vlissides, 1994]

Patrón Command

Propósito:

Encapsular una petición en un objeto, de manera que se puedan parametrizar los clientes con distintas peticiones, encolarlas o llevar un registro de las mismas, pudiendo deshacer dichas operaciones si se desea. [Gamma, Helm, Johnson, & Vlissides, 1994]

Motivación:

En ocasiones es necesario poder enviar peticiones a objetos sin conocer la operación solicitada o el receptor de la misma. Por ejemplo imaginar un conjunto de herramientas de cualquier interfaz de usuario, en el que se suelen encontrar botones o menús que lanzan una petición como respuesta a una entrada del usuario, pero estos elementos no se encargan de la implementación de esta solicitud, ya que sólo las aplicaciones que usen el conjunto de herramientas conocerán lo que se debe hacer y sobre que objeto realizarlo. Por tanto desde el punto de vista de un diseñador de conjuntos de herramientas no conoceríamos ni el contenido de la operación (o operaciones) a llevar a cabo tras una petición ni el receptor de ésta. [Gamma, Helm, Johnson, & Vlissides, 1994]

Estructura y participantes [Gamma, Helm, Johnson, & Vlissides, 1994]:

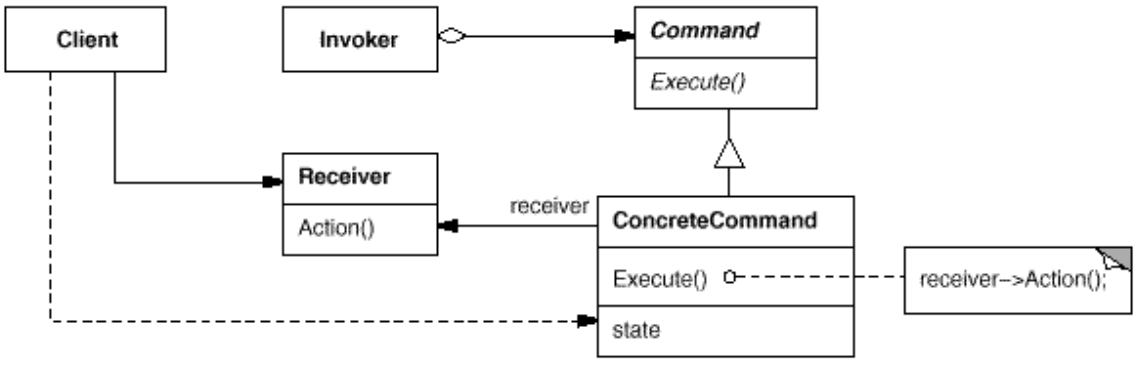


Figura 1. Estructura patrón Command.

- **Command (comando abstracto)**: declara una interfaz para ejecutar una operación. En caso de permitir deshacer comandos también debe añadir operación para ello.
- **ConcreteCommand (comando concreto)**: actúa de enlace entre el receptor y una acción, e implementa la ejecución del comando invocando la operación(es) correspondiente del receptor. Si se permite deshacer comandos debe guardar el estado antes de invocar al ejecutor de la acción.
- **Client (cliente)**: Crea un comando concreto (**ConcreteCommand)** y le asigna el receptor (**Receiver**).
- **Invoker (invocador)**: contiene el comando asociado a la petición.
- **Receiver (receptor)**: sabe cómo realizar las operaciones asociadas a una petición.

Utilidad [Gamma, Helm, Johnson, & Vlissides, 1994]:

- Permite parametrizar objetos con una acción.
- Especificar, encolar y ejecutar peticiones en distintos momentos, ya que un objeto de comando tiene un tiempo de vida independiente del de la petición original.
- Provee los medios necesarios para deshacer operaciones.
- Facilita la creación de un log de cambios para que estos puedan ser restaurados en caso de que el sistema falle. Para ello la interfaz de los comandos debe declarar métodos para guardar y cargar los comandos de disco.
- Estructurar un sistema en torno a operaciones de alto nivel constituidas sobre operaciones básicas. Los comandos tienen una interfaz en común que permite invocar a todas las acciones del mismo modo y además permite la incorporación de nuevas acciones de forma muy sencilla.

Patrón Chain of Responsibility

Propósito:

Evitar acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a una petición. Encadenar los receptores y pasar la petición a lo largo de esa cadena hasta que un objeto sea capaz de manejarla. [Gamma, Helm, Johnson, & Vlissides, 1994]

Motivación:

Suponer un servicio de ayuda sensible al contexto de una interfaz gráfica. El usuario puede pulsar con el ratón sobre cualquier zona de la interfaz para obtener la ayuda, cuyo contenido dependerá de esa zona y del contexto, es decir, no prestará la misma ayuda un botón de un cuadro de diálogo, que uno similar de la pantalla principal. Además en caso de no existir ayuda para el elemento sobre el que se ha pulsado, se debe mostrar la del contexto más inmediato que disponga de ella.
Por ello surge un problema si se da el caso en el que el objeto que finalmente proporciona la ayuda, no conoce al objeto que la solicita. Para solucionarlo se

necesitaría desacoplar el botón que solicita la ayuda de los objetos que puedan proveerla. Precisamente la idea del patrón Chain of Responsibility es ésta, desacoplar a los emisores y los receptores dando la oportunidad de responder a una petición a varios objetos, de tal manera que la petición pasa por la cadena formada por estos objetos hasta que uno de ellos es capaz de procesarla. [Gamma, Helm, Johnson, & Vlissides, 1994]

Estructura y participantes [Gamma, Helm, Johnson, & Vlissides, 1994]:

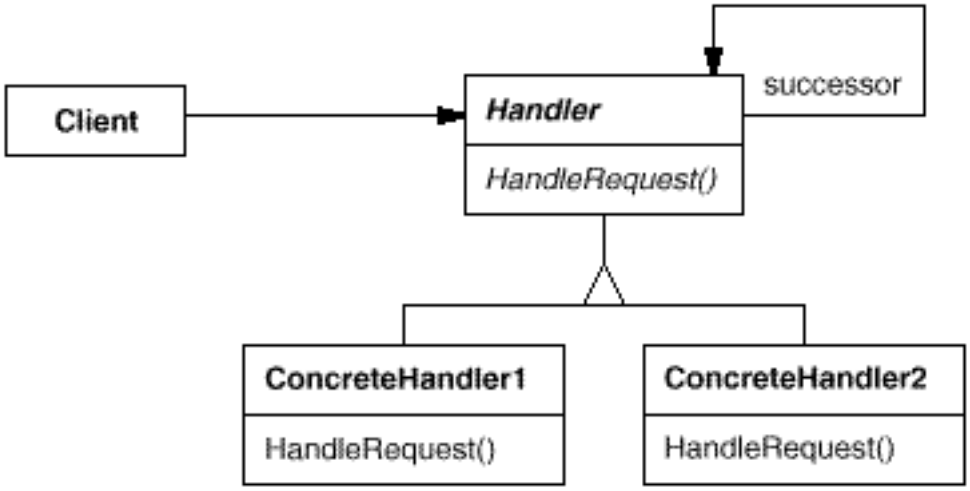


Figura 2. Estructura patrón Chain of Responsibility.

- ***Handler*** **(interfaz manejador)**: define una interfaz para procesar las peticiones y opcionalmente implementa el enlace con el sucesor.
- **ConcreteHandler (manejador concreto)**:maneja las peticiones de las que es responsable. Si puede procesarlas lo hace, si no envía la petición a su sucesor.
- **Client (cliente)**: envía la petición a un **ConcreteHandler** de la cadena.

Utilidad [Gamma, Helm, Johnson, & Vlissides, 1994]:

- En los casos en los que más de un objeto puede encargarse de una petición, pero no se conoce el encargado a priori, sino que debería determinarse automáticamente.
- También cuando se desea enviar una petición a un objeto de varios sin especificar a cuál explícitamente.
- Cuando el conjunto de objetos que pueden manejar una petición debería ser especificado dinámicamente.

Patrón Visitor

Propósito:

Representa una operación que se realiza sobre los elementos de la estructura de un objeto. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera. [Gamma, Helm, Johnson, & Vlissides, 1994]

Motivación:

Suponer un compilador que representa programas como árboles con sintaxis abstracta. Se necesitan operaciones sobre el árbol sintáctico tales como comprobación de tipos, generación de código… Estas operaciones necesitarán tratar los distintos tipos de nodos del árbol sintáctico (nodos de asignaciones, de variables, de expresiones…) de manera diferente. [Gamma, Helm, Johnson, & Vlissides, 1994]

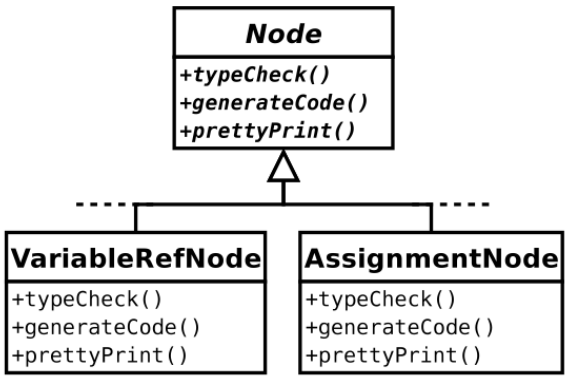


Figura 3. Ejemplo tipos de nodo de un compilador.

El principal problema de esto, es que al distribuir todos los tipos de operaciones por varias clases de nodos el sistema se vuelve más difícil de entender, cambiar y modificar. Una solución consiste en agrupar las operaciones relacionadas de cada clase en un objeto (Visitor) y pasar como argumento los elementos del árbol sintáctico. Cuando un elemento acepta al visitante, le envía una petición de su clase pasándose a sí mismo como argumento.
Por último, para poder permitir más de un visitante se defina una superclase abstracta con una operación por cada tipo de nodo. [Gamma, Helm, Johnson, & Vlissides, 1994]

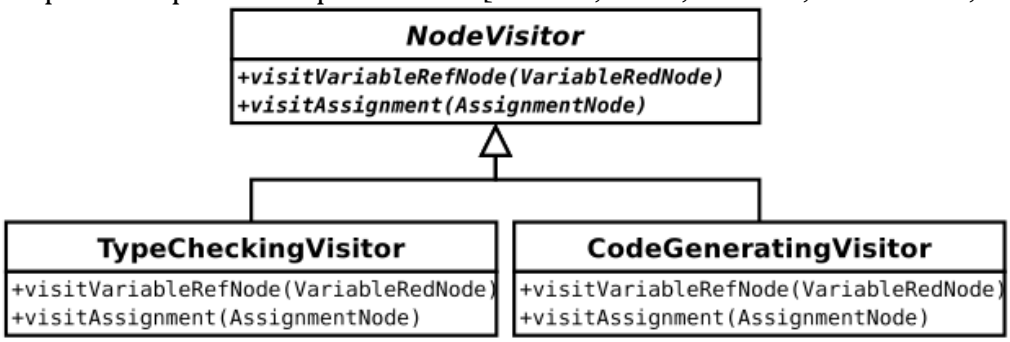


Figura 4. Ejemplo agrupación operaciones compilador.

Estructura y participantes [Gamma, Helm, Johnson, & Vlissides, 1994]:

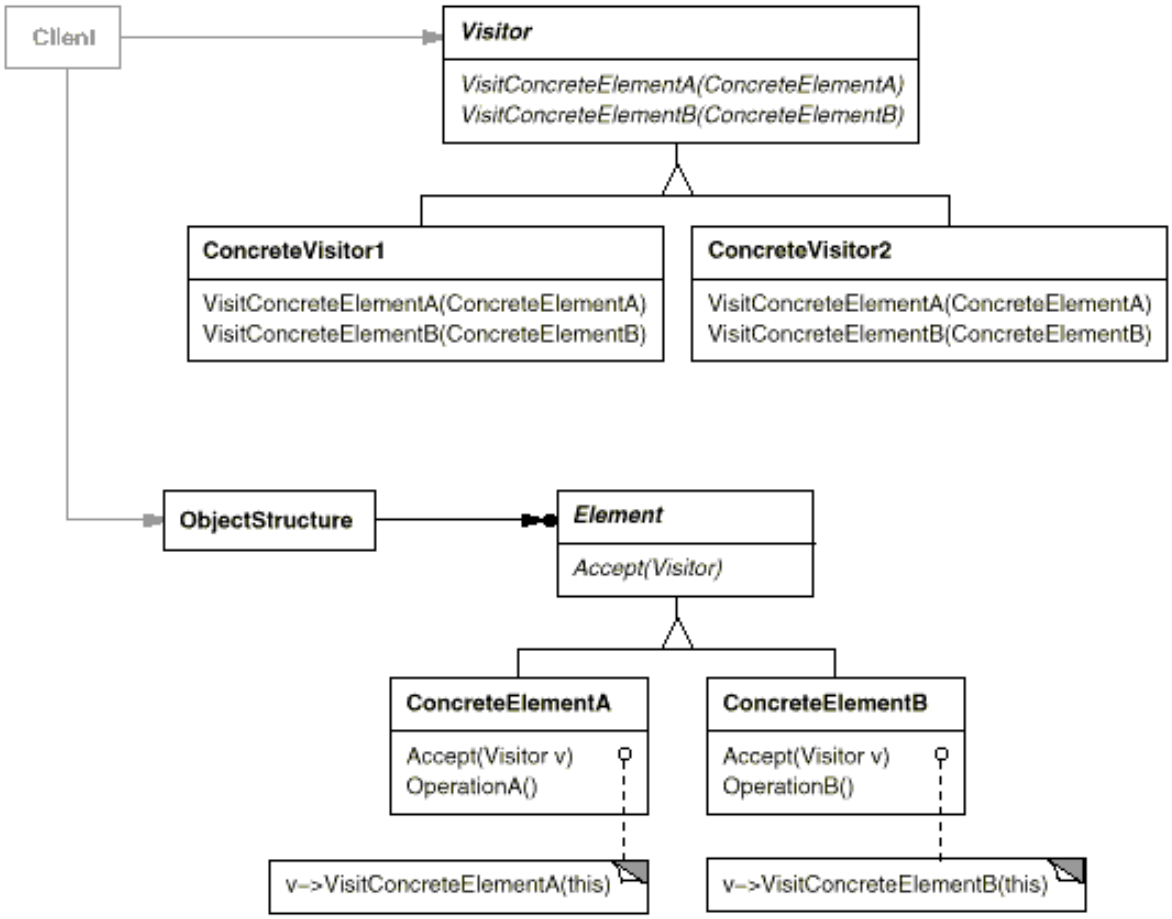


Figura 5. Estructura patrón Visitor.

- **Visitor**: declara una operación de visita por cada clase de **ConcreteElement** en la estructura de objetos, que incluye al propio objeto visitado.
- **ConcreteVisitor**: Implementa las operaciones declaradas por **Visitor**, las cuales representan los distintos fragmentos que componen la labor global del **ConcreteVisitor.** Además suele almacenar resultados en un estado local.
- **Element**: define una operación de Accept que lleva un **Visitor** como argumento.
- **ConcreteElement**: implementa la operación de Accept.
- **ObjectStructure**: gestiona la estructura de objetos y es capaz de enumerar los **elementos**. Es capaz de proporcionar una interfaz de alto nivel para permitir al **Visitor** visitar sus elementos. Puede ser una colección o un compuesto (patrón Composite).

Utilidad [Gamma, Helm, Johnson, & Vlissides, 1994]:

- Una estructura de objetos contiene muchas clases de objetos con interfaces distintas, y se desea realizar operaciones sobre ellos que dependan de sus clases concretas.
- Es necesario realizar muchas operaciones distintas en los objetos de una estructura y se quiere evitar "contaminar" las clases con estas operaciones.
- Cuando las clases que definen la estructura de objetos no cambian, pero sí surgen nuevas operaciones sobre ellas.

Patrón Strategy

Propósito:

Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Permite que un algoritmo sea modificado sin afectar al cliente. [Gamma, Helm, Johnson, & Vlissides, 1994]

Motivación [Gamma, Helm, Johnson, & Vlissides, 1994]:

Existen muchos algoritmos para dividir un flujo de texto en líneas; codificar estos algoritmos en las clases que los necesitan no es deseable por los siguientes motivos:

- Los clientes se vuelven más complejos, aumentando en tamaño y en dificultad de mantenimiento.
- Diferentes algoritmos serán apropiados en distintos momentos.
- Es más difícil añadir nuevos algoritmos y modificar los existentes, si están integrados en los clientes.

La solución sería definir clases que encapsulen los algoritmos.

Estructura y participantes [Gamma, Helm, Johnson, & Vlissides, 1994]:

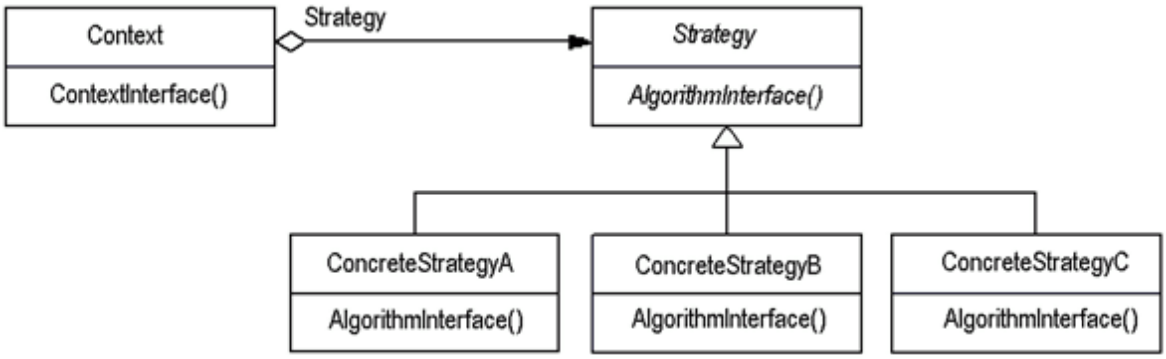


Figura 6. Estructura patrón Strategy.

- ***Strategy***: Declara una interfaz común para todos los algoritmos soportados. El **contexto** usa esta interfaz para llamar al algoritmo definido por **ConcreteStrategy.**
- **ConcreteStrategy**: Implementa el algoritmo usando la interfaz **Strategy**.
- **Context**: Está configurado con un objeto **ConcreteStrategy,** pero mantiene una referencia a un objeto ***Strategy***. Además puede definir una interfaz que permita a la estrategia acceder a los datos del contexto.

Utilidad [Gamma, Helm, Johnson, & Vlissides, 1994]:

- Muchas clases relacionadas sólo se distinguen por su comportamiento. Las estrategias permiten configurar una clase con uno de entre varios comportamientos.
- En el caso de que se necesiten diferentes variantes de un algoritmo.
- Cuando un algoritmo usa datos que el cliente no tiene por qué conocer.
- Si una clase define muchos comportamientos diferentes y éstos aparecen en forma condicional en sus operaciones, las ramas de cada condición pueden ser remplazadas por distintas estrategias.

Programación orientada a aspectos

La Programación Orientada a Aspectos es un paradigma de programación relativamente reciente, que trata de mejorar la modularización de las aplicaciones y posibilitar una mejor separación de competencias, capturando las mismas en entidades bien definidas para cada uno de los casos. Permitiendo razonar mejor sobre las competencias, eliminando la dispersión de código y produciendo implementaciones más comprensibles, adaptables y reutilizables. [Paz Rojas, 2007]

Según la definición de Gregor Kiczales, profesor de *University of British Columbia* en Canadá, pionero en el campo de la programación orientada a aspectos, que ha dirigido el equipo de desarrollo de la extensión de Java **AspectJ**, un aspecto se puede ver como: "*Una unidad modular que se dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa".* [Paz Rojas, 2007]

En la programación orientada a aspectos se establece que en las clases se implementa la funcionalidad principal de una aplicación, mientras que los aspectos capturan conceptos técnicos tales como la persistencia, la gestión de errores o la comunicación de procesos. [Paz Rojas, 2007]
Para ello un programa debe disponer de :

- Un **lenguaje para definir la funcionalidad básica**, que suele ser de propósito general, por ejemplo C++ o Java.
- **Uno o más lenguajes de aspectos**, para dar forma a los mismos, por ejemplo

AspectJ, en el que se programan de forma muy parecida a las clases.
- **Un tejedor de aspectos** encargado de la combinación de los lenguajes, que puede hacerse tanto en tiempo de compilación como de ejecución.

Entre los objetivos más destacables de la POA se encuentran el de separar conceptos y minimizar las dependencias entre ellos. De la consecución de éstos se pueden obtener las siguientes ventajas [Paz Rojas, 2007]:

- Código con mayor claridad, más natural y más reducido.
- Mayor facilidad para razonar sobre las materias, que se encuentran separadas y con una dependencia mínima.
- Más facilidad de depuración y modificación del código.
- Las modificaciones en la definición de una materia tienen un impacto mínimo en las otras.
- Se consigue un código más reutilizable, capaz de acoplarse y desacoplarse cuando sea necesario.

Reflexión y metaprogramación.

La meta-programación es el proceso por el cual se escriben programas que son capaces de manipular o generar otros programas como datos, o que son capaces de realizar parte del trabajo que se haría en tiempo de ejecución, en tiempo de compilación.
El término de reflexión puede verse como un caso particular de meta-programación de un programa consigo mismo, y que hace referencia a la capacidad de un programa para inspeccionar y manipular su estado (estructura o comportamiento) en tiempo de ejecución. [Universidad Tecnológica Nacional, Facultad Regional Buenos Aires] [Tanter, 2002]
Para ello el lenguaje de programación debe ser capaz de conservar ese estado en forma de meta-información, para que permanezca una vez se haya realizado la compilación.

En ocasiones se puede emplear reflexión en tiempo de compilación, a modo de pre-procesamiento y únicamente pudiendo manipular el contenido estático [Tanter, 2002].

La reflexión se utiliza con mayor frecuencia en lenguajes de alto nivel como C#, Smalltalk y Java, que es el que se utiliza durante el desarrollo del proyecto y que permite entre otras funciones reflexivas, inspeccionar clases, interfaces, campos y métodos sin conocer sus nombres en tiempo de compilación, crear instancias de las clases, e incluso manipular atributos e invocar métodos de las clases. [Tanter, 2002]
En muchas ocasiones se utilizan estas herramientas para modificar el comportamiento de un programa de forma dinámica, adaptándose a distintas situaciones sin la necesidad de tener definidas todas ellas de manera estática.

El principal inconveniente de la reflexión es la pérdida de rendimiento, debido a que

en cada paso realizado de manera reflexiva debe ser validado, y por ejemplo en lenguajes como Java al resolverse dinámicamente los tipos, es imposible que se puedan aplicar las optimizaciones que ofrece la "Java Virtual Machine".

ORM (Object-Relational Mapping ó Mapeo Objeto-Relacional)

Es una técnica de programación que busca conectar los paradigmas de la programación orientada a objetos (POO) y el modelo de bases de datos relacional [Kuaté, Bauer, King, & Harris, 2009], utilizados simultáneamente en la mayoría de las aplicaciones de hoy en día.

Las principales ventajas que puede ofrecer un ORM son [Kuaté, Bauer, King, & Harris, 2009] [López Rodriguez] :

- Aumentar la transparencia a la hora de realizar la persistencia de datos, en la medida de lo posible.
  Reduciendo por tanto el esfuerzo del programador a la hora de manejar la persistencia de datos, la cual implicaría mucha dedicación en la componente de bases de datos, teniendo que realizar gran cantidad de código repetitivo. De esta manera aumenta la productividad, permitiendo que ese tiempo que se ha reducido pueda dedicarse a otros componentes de la aplicación.
- Abstracción e independencia del motor de base de datos. Ya que son capaces de generar consultas para transformar los objetos en registros y viceversa, adaptándose a distintos motores de bases de datos (MySQL, PostgreSQL, Oracle…).
- Permiten la construcción de consultas a bases de datos mediante un lenguaje orientado a objetos, dejando de lado la semántica SQL.
- Mejora la mantenibilidad de la aplicación, por un lado porque reduce el número de líneas de código (dando mayor protagonismo a la lógica de negocio de la aplicación), y porque además el código generado por un ORM suele estar completamente testeado, lo cual evita tener que preocuparse mucho menos por esa parte de la aplicación a la hora de comprobar su correcto funcionamiento, ya que el principal fallo que podría producirse es que no se estuviera realizando la operación deseada, pero nunca un fallo sintáctico a la hora de formar las consultas. Hace que sea más fácil reestructurar el esquema de base de datos o el modelo sin afectar el uso de los objetos por parte de la aplicación.

Asimismo, un ORM tiene que hacer frente a una serie de problemas y tiene diversas desventajas:

- Al tener que llevarse a cabo la transformación entre dos paradigmas surgen una serie de problemas intrínsecos (conocidos comúnmente como object/relational impedance mismatches) [Kuaté, Bauer, King, & Harris, 2009] [Keith & Schincariol, 2009]:
  - **Granularidad**.
    El paradigma Orientado a Objetos (OO) ofrece distintos niveles de granularidad, desde clases elementales (por ejemplo String), clases

simples, hasta clases compuestas por otras e incluso éstas a su vez por otras. En cambio en el paradigma relacional sólo se dispone de dos niveles de granularidad, que son tablas y columnas. Por tanto este es un problema que el ORM deberá ser capaz de solucionar.

- **Herencia y polimorfismo**.
  Ambos son conceptos que disponen de soporte en POO pero que requieren una adaptación en el modelo relacional. Por ejemplo, suponer una clase abstracta de la que heredan dos concretas. Existen distintas posibilidades para adaptarlo al modelo relacional como por ejemplo una tabla para cada una de las clases concretas, una única tabla con alguna columna discriminante, o una tabla por cada clase, y dependiendo de la situación puede interesar más una solución u otra.
- **Identidad.**
  La identidad en una base de datos se comprueba a través de las claves primarias de tablas, pero en POO además de una posible igualdad por valor, existe la denominada identidad por objeto, que consiste en la comparación de la posición de memoria de los objetos, es decir, dos objetos son idénticos si apuntan a la misma dirección de memoria.
- **Asociaciones.**
  En POO las asociaciones son direccionales, es decir, si se quiere que dos objetos estén asociados bidireccionalmente, se deberá definir la asociación en ambos objetos. En cambio, en el paradigma relacional las asociaciones (creadas mediante claves foráneas), no son direccionales y permiten realizar asociaciones arbitrarias entre los datos, por ejemplo mediante joins.
- **Navegación.**
  El modo de acceder a los datos en POO es diferente al del modelo relacional. En el primero podemos navegar de objeto en objeto mediante sus métodos de acceso hasta alcanzar el dato deseado, en cambio, en el segundo, navegar de una entidad a otra requiere uno o más joins.
  Uno de los principales intereses es tratar reducir el número de joins y de consultas para mejorar la eficiencia de la aplicación. Para ello es conveniente conocer las distintas entidades que vamos a necesitar recorrer antes de comenzar el camino y poder decidir así la alternativa más eficiente.

- Requiere dedicación al aprendizaje del uso del ORM para poder explotar sus capacidades al máximo.
- Al añadir una nueva capa intermedia a la aplicación puede reducirse el rendimiento de la misma, sobre todo si se trata de aplicaciones muy complejas y con gran carga.
- Al crear un lenguaje de consulta que se abstrae de los distintos motores de bases de datos, se produce un estándar que en ocasiones puede dificultar la posibilidad de acceder a funcionalidades avanzadas y propias de un motor

específico que pueden ser útiles.

- Para evitar la carga de mucha información de la base de datos, los ORMs suelen detectar cuando se trata de acceder a un objeto relacionado y en ese momento carga la información de éste; el problema surge cuando se trata de serializar un objeto, ya que el sistema no es capaz de distinguir hasta dónde debe llegar al recorrer cada una de las propiedades del objeto y su contenido interno; por consecuencia, serializar un objeto puede tener como efecto colateral que se incluya en la serialización mucha más información de la necesaria.

Java Bean Validation

Definición de **Java Bean** [Sun Microsystems]:
Es un componente de software reutilizable cuyas características principales son:
- Permiten la introspección, de manera que se pueda analizar cómo funcionan.
- Permiten modificar su apariencia y/o comportamiento.
- Son capaces de comunicarse mediante eventos.
- Tienen propiedades, accesibles de manera programática a través de llamadas a sus getters y setters (métodos de acceso).
- Tienen la posibilidad de hacerse persistentes.

Java Bean Validation es una especificación aprobada en los marcos de JCP (Java Community Process) el 16 de noviembre de 2009, siendo aceptado como parte de la especificación de Java EE 6, que ofrece la posibilidad de llevar a cabo la validación de JavaBeans a través de metadatos, que actúan como restricciones sobre objetos, en forma de anotaciones [Bernard].

Estas anotaciones se podrán aplicar sobre tipos, campos, métodos, constructores, parámetros, o sobre otra restricción en forma de anotación (composición de restricciones). Además se podrán definir restricciones nuevas [Bernard].

Presentación de Uaithne

Uaithne se podría definir como un híbrido entre framework y generador de código.

Es un **framework** de Java porque conforma un conjunto de disciplinas que se han conseguido estructurar en forma de clases para su posterior reutilización. Define una arquitectura de backend que se marca como objetivos:

- Facilitar el crecimiento de los sistemas.
- Permitir modificaciones en los sistemas sin grandes costes.
- Dar soporte y hacer más sencilla la separación de competencias tal y cómo pretende la programación orientada a aspectos (**POA**).

Además es un **generador** de código que ahorra muchas líneas de código repetitivo, algunas de ellas consecuencia de la verbosidad de la arquitectura exigida por el framework, y otras muchas relacionadas con la implementación de acceso a base de datos, que el generador permite activar de manera opcional, utilizando MyBatis[1].

La arquitectura que define, facilita la creación del **bus** de una aplicación encargado de comunicar las distintas piezas o componentes de la aplicación para construir cada una de las capas en que se constituye, tal como si de un juego de LEGO se tratase, permitiendo configurar, reconfigurar y reordenar sus componentes fácilmente, y permitiendo la separación de competencias mediante la implementación de interceptores[2] de operaciones que se pueden incorporar fácilmente al bus.

Con esto se consigue que la aplicación pueda crecer y ser modificada sin que suponga un gran esfuerzo, y con ayuda del generador se consigue ahorrar muchas líneas de código, mejorando así la productividad.

Uaithne distingue dos niveles lógicos, uno que corresponde con los **módulos de operaciones** (o Executors) y otro superior que consiste en **agrupaciones** de estos módulos (ExecutorGroups).

Las agrupaciones de módulos (o capas de la aplicación) reciben una operación y son capaces de retornar el resultado de la operación, para ello deben saber a qué módulo delegar la ejecución de la operación y ahí es donde entra en juego el patrón de comportamiento **Strategy.**

Los módulos a su vez pueden necesitar delegar la ejecución a otro módulo, de ahí que ambos implementen la interfaz **Executor** y sea posible aplicar el patrón **Chain of**

---

[1] MyBatis es un framework de persistencia que soporta SQL, procedimientos almacenados y mapeos avanzados. MyBatis elimina casi todo el código JDBC, el establecimiento manual de los parámetros y la obtención de resultados. MyBatis puede configurarse con XML o anotaciones y permite mapear mapas y POJOs (Plain Old Java Objects) con registros de base de datos. [MyBatis]

[2] Un interceptor es un elemento que se interpone en el flujo de ejecución de la operación permitiendo modificar su comportamiento; por ejemplo, se puede decir que un trigger de base de datos actúa como un interceptor de las operaciones que se realizan sobre una tabla.

Responsibility. Esto también puede ocurrir entre las agrupaciones, que implementan la interfaz **ExecutorGroup;** se puede dar el caso de que la operación llegue a una capa (agrupación) pero su implementación no se encuentre dentro de ella sino en una capa inferior y por tanto deba ser delegada a la siguiente capa.

Dentro del nivel lógico más interno se encuentran las **operaciones,** que Uaithne las define como acciones en forma de objetos (interfaz **Operation**) utilizando el patrón **Command.**

Pero dado que interesa que una operación pueda tener distintas implementaciones (por ejemplo dependiendo de la capa en la que se encuentre), Uaithne genera interfaces de **Visitor,** haciendo separaciones lógicas entre cada módulo, para evitar tener todas las operaciones en una misma interfaz; por ello estas interfaces generadas extienden la clase **Executor**.

Para poder generar las implementaciones de los accesos a bases de datos ofrece la posibilidad de crear entidades, que representan vistas de datos y también facilita la creación de los distintos tipos de operaciones posibles contra bases de datos mediante un sistema de anotaciones propias que se colocan sobre clases y atributos.
Puede decirse que en este aspecto trata de explotar algunas de las ventajas que ofrecen los ORMs, como disminuir la escritura de del código repetitivo, aumentar la productividad, la abstracción del motor de bases de datos, etc. Pero en ningún momento intenta ser un ORM, ya que está destinado a usos dónde un ORM no es apropiado, por ejemplo un sistema web dónde es necesario serializar la información para enviársela al navegador.

Diseño y desarrollo

Diseño e implantación de pruebas unitarias.

Se ha llevado a cabo el diseño y desarrollo de pruebas unitarias, mediante el framework de automatización de pruebas unitarias para Java JUnit, en su versión 4.10, centrándose en aquellas clases del generador de Uaithne más críticas a la hora de llevar a cabo la evolución del proyecto, para que de esta manera cada vez que se hagan modificaciones se pueda tener un control que evite cambios no deseados o resultados no esperados, lo cual afectaría gravemente a aquellos proyectos que se estén desarrollando con la ayuda de Uaithne.

Estas clases han sido:
ClassTemplate : genera las plantillas de las clases, es decir, todo aquello que engloba el contenido de la clase (anotaciones de clase, comienzo y final).
WithFieldsTemplate : extiende ClassTemplate y contiene métodos para la generación de los argumentos cuando aparecen en las invocaciones de los métodos y cuando se lleva cabo la declaración de estos últimos.
PojoTemplate : extiende WithFieldsTemplate y se encarga de generar los componentes característicos de un POJO como son atributos privados, getters y setters, con su respectiva documentación, además de otros métodos comunes como ToString, Equals, HashCode y las anotaciones de los elementos de la clase.
EntityTemplate : extiende PojoTemplate y contiene los métodos para generar todo el contenido de la clase utilizando los métodos que definen las clases anteriores y la generación de constructores especificada en esta misma.

SqlGenerator : clase que implementa la interfaz **QueryGenerator** (declara métodos para la generación de consultas y acceder y modificar la configuración de esta generación). En esta clase únicamente se crea el atributo que almacenará la configuración, y algunos métodos auxiliares que se usarán en las clases que extiendan de ella a la hora de generar las consultas.
SqlQueryGenerator : extiende de la clase anterior y se puede considerar la clase más crítica ya que es la encargada de generar las consultas en sql. Tiene una extensión de aproximadamente 2000 líneas de código, siendo con diferencia la de mayor tamaño dentro del generador.

MyBatisSqlQueryGenerator : extiende SqlQueryGenerator y se encarga de adaptar esta clase para la generación de las consultas sql utilizando MyBatis.

Los test generados para estas clases se han denominado con el nombre de la clase y el sufijo Test.

La primera clase que se ha probado ha sido **ClassTemplate** con el fin de continuar con las demás clases que extienden de ella en orden.

Dado que en esta clase se lleva a cabo la generación de todo aquello que engloba el contenido de una clase o interfaz, como puede ser añadir los imports, comentarios, escribir las implementaciones o superclases, anotación "deprecated" de la clase, etc. es necesario utilizar objetos simulados que tengan la forma de las distintas clases o interfaces que se pueden formar para poder llevar a cabo las pruebas comparando estos objetos simulados que representan el resultado esperado, con la implementación que se elabora de la clase a probar.
Como ejemplo ilustrativo observar el caso en el que se prueba la creación de una clase normal cuya característica es que dispone de documentación.

```
@Test
public void testWriteSimpleClassWithDoc() throws Exception {
    Appendable appender = new StringBuilder();
    ClassTemplate instance = new ClassTemplateImpl();
    instance.setClassName("ClassTemplateTestResult1");
    instance.setPackageName("org.uaithne.generator.templates");
    instance.setDocumentation(new String[]{"Documentation", "of the class"});
    instance.write(appender);
    String expected = loadResult("ClassTemplateTestResult1");
    Assert.assertEquals(expected, appender.toString());
}
```

La clase que simula la salida esperada es ClassTemplateTestResult1 y se define de la siguiente manera:

```
package org.uaithne.generator.templates;

/**
 * Documentation
 * of the class
*/
public class ClassTemplateTestResult1 {



}
```

A continuación se han llevado a cabo los test de las clases **WithFieldsTemplate, PojoTemplate** y **EntityTemplate**, en ese orden; pero en este caso no ha sido necesario utilizar objetos que simulasen el resultado ya que todas las funciones lo que hacían era escribir en un objeto de tipo Appendable el código a generar. Por tanto

para llevar a cabo las diferentes pruebas de cada método se han simulado los distintos caminos que podían seguir los métodos para lograr la máxima cobertura, y se iba comparando cada camino escrito en el Appendable con el String esperado.

La clase **SqlGenerator** no requiere ninguna mención especial, se han probado todos sus métodos buscando nuevamente la máxima cobertura posible.

SqlQueryGenerator es la clase más crucial del generador dónde se han realizado pruebas. En ella se lleva a cabo la generación en sql de consultas, por tanto debe soportar la mayoría de combinaciones posibles que se puedan imaginar en este lenguaje para acabar formando una consulta sintácticamente correcta. De ahí la importancia de llevar a cabo pruebas sobre ella, ya que si se producen evoluciones sobre el generador las pruebas indicarán si se está rompiendo lo que había implementado hasta ahora y que funcionaba en los proyectos en los que se encuentra Uaithne acoplado actualmente, evitando sorpresas inesperadas.

La clase de pruebas que se ha elaborado para esta clase ha alcanzado las 3850 líneas de código (aproximadamente el doble que la clase original). Para poder realizar estas pruebas han tenido que crearse varios objetos simulados entre los que merece la pena destacar los siguientes:

- **MessageContent**, **ProcessingEnviromentImpl** y **MessagerImpl.** Los tres están relacionados. El generador utiliza un objeto de tipo **ProcessingEnvironment** de Java para mostrar errores o warnings durante la generación; para simular este objeto se crea una implementación de la interfaz **ProcessingEnvironment** denominada **ProcessingEnviromentImpl.** Este objeto provee otro de tipo **Messager** (también de Java), que es el verdadero encargado de mostrar los mensajes; por tanto ha sido necesario también crear una implementación (**MessagerImpl**) que en vez de imprimir el mensaje lo almacene en una nueva clase llamada **MessageContent** y asignarle esta implementación de **Messager** a **ProcessingEnviromentImpl.**
  MessageContent contiene los tres atributos que recibe como parámetro la función de impresión del Messager, que son: un String con el **mensaje**, el **tipo** del mensaje (Diagnostic.Kind) y el **elemento** que produce el mensaje (Element). Además se sobrescriben los métodos **equals** y **hashCode** de esta clase para comparar dos **MessageContent** campo a campo.

  De esta manera se puede comprobar mediante un Assert si el **MessageContent** que devuelve el método que se prueba coincide con uno esperado creado manualmente.

- **TestCustomSqlQuery** es un objeto simulado que implementa la interfaz de la anotación **@CustomSqlQuery** (utilizada para poder indicar partes como el select, groupby, where, etc. de una consulta manualmente). Se ha definido el objeto de tal manera que cada uno de sus métodos devuelva el valor de un

atributo público, de forma que se puedan configurar los atributos para que sus métodos asociados devuelvan un determinado resultado.
De esta manera se pueden probar muchos caminos lógicos de métodos que utilizan o condicionan su comportamiento en función de este objeto.

- En muchas ocasiones es necesario simular entidades u operaciones con unos determinados tipos de campos (identificadores, marcas de borrado, opcionales, etc.) para ello según se ha necesitado se han ido declarando métodos que crean instancias de **EntityInfo** u **OperationInfo** configuradas según lo requerido para las pruebas.

En **MyBatisSqlQueryGenerator** se prueban los métodos que llevan a cabo la especificación para adaptar el sql a XML de MyBatis. No requiere la utilización de objetos simulados tan complejos como en el caso anterior.

Generación de documentación de uso de la herramienta.

Se ha llevado a cabo la elaboración del documento "Introducción práctica a Uaithne" (disponible en el Anexo A) que presenta una serie de conocimientos necesarios para poder utilizar la herramienta Uaithne en el desarrollo ágil del backend de aplicaciones Java; por tanto el nivel de conocimiento requerido para entender sin problemas el documento es el de alguien que tenga al menos nociones básicas sobre el desarrollo de este tipo de backends.

El objetivo del documento es que tras una primera lectura el desarrollador sea capaz de comprender el funcionamiento de la herramienta, la utilidad que puede ofrecer a sus proyectos, y posteriormente ayudarle a explotar todas sus capacidades y servir de apoyo como documento de consulta en caso de surgir dudas durante la utilización de la herramienta.

El documento se ha elaborado en torno a casos prácticos para que el lector sea capaz de enfocar los conceptos en ejemplos reales, sencillos y comunes durante el desarrollo de aplicaciones, facilitando y mejorando el tiempo de aprendizaje de la herramienta.

A continuación se mencionan los principales temas que trata el documento sin entrar en gran detalle.

- **Entidades y módulos de operaciones.**
  Se describe la forma de crear entidades (representaciones de objetos o conceptos del mundo real) mediante anotaciones y distinguiendo entre los dos tipos que ofrece la herramienta **@Entity** y **@EntityView.** La diferencia que

existe es que la primera, si se sitúa en un módulo de operaciones, genera una serie de operaciones (consulta, inserción, actualización y borrado) por defecto y la segunda no.
Además se indica la posibilidad de realizar agrupaciones lógicas mediante los módulos de operaciones (**@OperationModule**), que permiten señalar que la lógica de acceso a base de datos de las operaciones de ese módulo se genere utilizando la herramienta MyBatis marcando el módulo con la anotación **@MyBatisMapper**.

- **Operaciones.**
  Las operaciones representan las acciones que lleva a cabo la aplicación, son implementadas como **Comandos,** a modo de objetos que encapsulan las acciones y tienen una interfaz común para poder invocar estas acciones y facilitar la creación de nuevas operaciones.
  Se explica al lector las múltiples posibilidades que existen para crear operaciones, tanto si son de acceso a base de datos de cualquier tipo (insert, select, update, delete, etc.), como definidas completamente por el usuario. Se comenta el proceso de elaboración de consultas tan simples como un select sobre una tabla, hasta las más complejas que se puedan imaginar. Durante el proceso se explica la gran variedad de anotaciones que permiten personalizar los parámetros y campos involucrados en las operaciones y que otorgan el control total al programador para crear la operación deseada de manera sencilla y sin necesidad de tener que repetir código.

- **Configuración del bus.**
  Uno de los aspectos más importantes de Uaithne es que facilita la creación de un bus que comunica las distintas capas de la aplicación mediante la conexión de componentes capacitados para alojar los distintos módulos (de operaciones) de cada capa.
  De forma que una operación pueda atravesar las capas de la aplicación que se deseen, ahorrando costosos procesos de comunicación entre capas en cada una de las operaciones, y el resultado recorrerá el camino inverso.
  En este apartado del documento se explica cómo establecer relaciones entre esos dos niveles para poder elaborar las distintas capas del bus de la aplicación hasta completarlo y cómo configurar los comportamientos más comunes que atañen a una operación, como pueden ser implementarla, interceptarla o implementarla utilizando recursos provenientes de otras operaciones.

Evolución de la funcionalidad actual.

Uaithne es un proyecto que se encuentra en una fase de evolución constante, debido a que es un software realmente reciente (comenzó a desarrollarse a mediados del 2011) y según se va acoplando a otros proyectos es posible ir viendo distintas necesidades de mejora y ampliación, aunque se podría decir que ya está acercándose a una fase mucho más estable.
A continuación se describen algunas de las funcionalidades que se han incorporado como trabajo de este proyecto.

Implementación de Herencia entre "EntityViews"

Motivación.

La motivación de implementar una herencia lógica entre entidades es la posibilidad de que dada una entidad, pueda aparecer otra más específica que contenga campos de la original (más general) sin la necesidad de volver a definir las peculiaridades de cada campo, sino que simplemente baste con referenciarlo. Lo que da lugar a una mejor reutilización del código y reducción de los problemas derivados de su duplicación.

Diseño.

Para proceder a realizar la implementación de esta funcionalidad se ha tomado como referencia la generación de operaciones asociadas a una entidad, ya que el procedimiento para relacionar una entidad con otra será el mismo:

- Indicar en el parámetro "related" de la anotación de entidad (@Entity o @EntityView) la entidad (clase) con la que están relacionadas.
- En la nueva entidad, aquellos campos que se quieran heredar darles el mismo nombre que tienen en la entidad con la que guardan relación.

A nivel de entidad, la entidad más específica tomará de la más general , en caso de que no se haya especificado, la **documentación** y todas las **anotaciones** de las entidades relacionadas, teniendo en cuenta que en caso de que se repita alguna anotación se tomará la que se encuentre en la entidad más específica.

El nombre de la tabla a la que hace referencia la entidad se obtendrá por orden de disponibilidad de:

1. La anotación **@MappedName** (se utiliza para definir el nombre real de la entidad en la base de datos) de la entidad más específica.
2. La anotación **@MappedName** de la entidad más general a la que se pueda llegar.
3. El nombre de la entidad más general a la que se pueda llegar.

A nivel de campos de la entidad, lo único que hay que hacer, en el caso de que un campo de la entidad específica se llame igual que alguno de la entidad relacionada, es asignar a la propiedad related del campo, el de la entidad relacionada.
El único caso en el que hay que tener especial atención es cuando el campo al que se hace referencia en la entidad con la que se relaciona es de tipo identificador; si es así

se debe marcar como identificador también en la entidad específica, ya que en el caso de relación operación-entidad no se comprueba.

Desarrollo.
La primera necesidad que ha surgido ha sido permitir a las anotaciones @Entity y @EntityView la posibilidad de especificar el parámetro "related". Ha sido tan simple como añadir a la interfaz de la anotación la siguiente línea:

```
Class<?> related() default Void.class;
```

De esta manera cuando se escriba la anotación se podrá asignar la entidad relacionada como una clase y en caso de que no se especifique tomará por defecto la clase Void, para que cuando la encuentre el procesador no la asocie con ninguna otra entidad.

Para poder almacenar ese "related" es necesario añadir un atributo a la entidad que haga referencia a otra, esto se hace dentro de la clase **EntityInfo**, aunque también es necesario de manera auxiliar un atributo que simplemente almacene el nombre de la clase con la que se relaciona debido a que el procesamiento de las entidades y de las operaciones de entrada del generador se realiza en dos pasos.

Prestando atención al procesamiento de entidades, que es el de interés en este caso, el primer paso es en el que se procesan todas las entidades (Entity y EntityView) y es donde en el caso de que una entidad contenga el parámetro related distinto de Void.class se almacena en el atributo auxiliar.
En el segundo, es en el que se lleva a cabo la generación, pero justo antes de comenzar este paso es cuando se lleva a cabo la combinación de entidades, es decir, cuando ya están todas las entidades formadas como elementos individuales se procede a combinar unas con otras en caso de que éstas tengan alguna entidad padre o algún "related". En el caso de tener algún "related" se busca en un HashMap que contiene todas las entidades asociadas a su nombre, la entidad ya procesada que corresponda con el nombre almacenado previamente en el atributo auxiliar. Esta entidad es la que se asigna al atributo related de la clase **EntityInfo**.
Tras ello ya se puede proceder a la generación de las clases correspondientes a las entidades.

Generación de integración con procedimientos almacenados.

Motivación.
En gran cantidad de proyectos se trabajan con procedimientos almacenados para acceder a la base datos en vez de especificar las consultas directamente, algunas de

las ventajas por las cuales se opta por esta opción son las siguientes [Microsoft]:

- Reduce el tráfico de red entre el cliente y el servidor, ya que el código de los comandos a ejecutar no se tienen que enviar por la red sino que está encapsulado dentro de los procedimientos almacenados y únicamente es necesario enviar la llamada al procedimiento deseado.
- Hay aplicaciones en las que varios usuarios y programas cliente pueden acceder a los objetos de la base de datos subyacentes. Se puede hacer que estos clientes accedan a la base de datos a través de procedimientos almacenados, que controlan que procesos y actividades se llevan a cabo, en vez de directamente a esos objetos y así controlar de manera mucho más determinista los accesos.
  Otra ventaja es que cuando se llama a un procedimiento a través de la red, sólo está visible la llamada que va a ejecutar el procedimiento, por tanto usuarios malintencionados no pueden ver los nombres de los objetos de base de datos y tablas, ni buscar datos críticos.
  Además el uso de parámetros de procedimientos ayuda a protegerse contra ataques por inyección de código SQL, dado que la entrada de los parámetros se trata como un literal y no como código ejecutable.
- Si en una aplicación el cliente llama a procedimientos almacenados y mantiene las operaciones de base de datos en la capa de datos, el nivel de aplicación permanece independiente y no tiene porqué tener conocimiento de los cambios realizados en los diseños o procesos de la base de datos.
- Mejora de rendimiento. Permite ejecutar lógica compleja (que requiera mucha interacción con los datos) dentro de la base de datos, que si tuviera que realizarse fuera de ella implicaría una mayor demora en la ejecución, debido a la necesidad de comunicación y transmisión de los datos a través de la red.

Además para poder agrupar estos procedimientos de forma lógica se ha decidido ofrecer también la posibilidad de generar paquetes.

Diseño.

Uaithne permite indicar que backend (sistema de gestión) de bases de datos es el utilizado por la aplicación, con el fin de que genere el código de las operaciones para el motor correspondiente.
Entre los distintos motores se pueden encontrar diferentes versiones de Oracle, SQL Server, Postgre SQL, Derby y también el estándar SQL 2008.

Para llevar a cabo la integración de los procedimientos de bases de datos se ha centrado la atención en el motor de Oracle, la razón es que gran parte de los proyectos en los que participa la empresa Delonia Software S.L. lo utilizan. Más concretamente se utilizará Oracle 10, por tanto aparecerán dos nuevas opciones de backend ORACLE_10_PROCEDURE y ORACLE_10_PACKAGE (para la generación de paquetes que agrupen los procedimientos).

Para poder llevar a cabo la generación e integración de los procedimientos almacenados es necesario conocer su estructura y la manera en la que son invocados. De tal forma que se pueda conseguir **generar un fichero .sql** en el que se encuentran los procedimientos correspondientes de las operaciones y trasladable fácilmente a la base de datos del sistema, y la ejecución de la operación consista en una **simple llamada al procedimiento**.

La estructura con la que se ha llevado a cabo la **implementación de los procedimientos almacenados** es la siguiente:

```
create or replace
procedure nombre_procedimiento[ (
   tipo_parámetro1 pnombre_parametro1 [,
   tipo_parámetro2 pnombre_parametro2...]
)]is
begin
   [código a ejecutar]
end nombre_procedimiento;
```

Donde **tipo_parámetroX** puede ser :

- **IN (entrada):**el parámetro puede ser referenciado por el procedimiento pero no se puede sobrescribir su valor.
- **OUT(salida):**el parámetro no puede ser referenciado por el procedimiento pero, en este caso, si se puede sobrescribir su valor.
- **IN OUT (entrada/salida):**el parámetro puede ser referenciado y modificado por el procedimiento.

La estructura de las **llamadas a los procedimientos** (utilizando MyBatis) es la siguiente:

```
call nombre_procedimiento[(
      #{nombre_parámetro1,jdbcType=tipo_jdbc1,mode=tipo_parámetro1,
            [,typeHandler= type_handler1]},
      #{nombre_parámetro2,jdbcType=tipo_jdbc2,mode=tipo_parámetro2
            [,typeHandler= type_handler2]...}
   )]
```

En el caso de **generación de paquetes** que contienen estos procedimientos la estructura será la siguiente:

1. Para el fichero .sql:

```
create or replace
package nombre_paquete as

[parte de especificación o cabeceras]
procedure nombre_procedimiento1[ (
      tipo_parámetro1 pnombre_parametro1 [,
```

```
        tipo_parámetro2 pnombre_parametro2...]
)];

procedure nombre_procedimiento2[ (
        tipo_parámetro1 pnombre_parametro1 [,
        tipo_parámetro2 pnombre_parametro2...]
)];

[...]

end nombre_paquete;
/

[cuerpo del paquete]
create or replace
package body nombre_paquete as

procedure nombre_procedimiento1[ (
   tipo_parámetro1 pnombre_parametro1 [,
   tipo_parámetro2 pnombre_parametro2...]
)]is
begin
    [código a ejecutar]
end nombre_procedimiento1;

procedure nombre_procedimiento2[ (
    tipo_parámetro1 pnombre_parametro1 [,
    tipo_parámetro2 pnombre_parametro2...]
)]is
begin
    [código a ejecutar]
end nombre_procedimiento2;

 [...]

end nombre_paquete;
/
```

2. Para las llamadas a los procedimientos:

```
call nombre_paquete.nombre_procedimiento[(
        #{nombre_parámetro1,jdbcType=tipo_jdbc1,mode=tipo_parámetro1,
                [,typeHandler= type_handler1]},
        #{nombre_parámetro2,jdbcType=tipo_jdbc2,mode=tipo_parámetro2
                [,typeHandler= type_handler2]...}
```

)]

Desarrollo.
Se comenzó llevando a cabo el desarrollo para poder generar las llamadas a procedimientos almacenados.

Para ello hay una clase abstracta llamada **SqlCallGenerator** que extiende de **SqlGenerator,** que como se ha mencionado en el apartado de pruebas unitarias implementa una interfaz que contiene los métodos que son necesarios sobrescribir para generar el código de los distintos tipos posibles de consultas a bases de datos. **SqlCallGenerator** provee la funcionalidad para generar sql en formato de llamada de procedimientos, que sirve tanto para generar la invocación como la definición de los mismos.

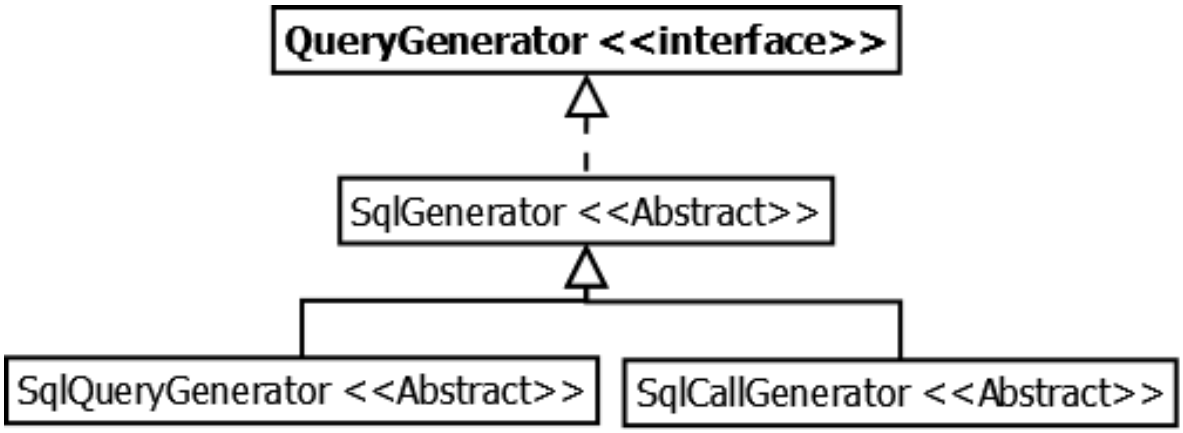


Figura 7. Interfaz QueryGenerator e implementaciones abstractas.

Como ya se ha comentado, en este caso lo que se desea es sustituir el código sql de esas consultas por la llamada a su correspondiente procedimiento almacenado.

A partir de **SqlCallGenerator** se crea otra clase abstracta que añade las modificaciones necesarias para adaptar las llamadas a MyBatis (**MyBatisSqlCallGenerator**) y por último la clase concreta para el motor de base de datos de Oracle **MyBatisOracleSqlCallGenerator.**

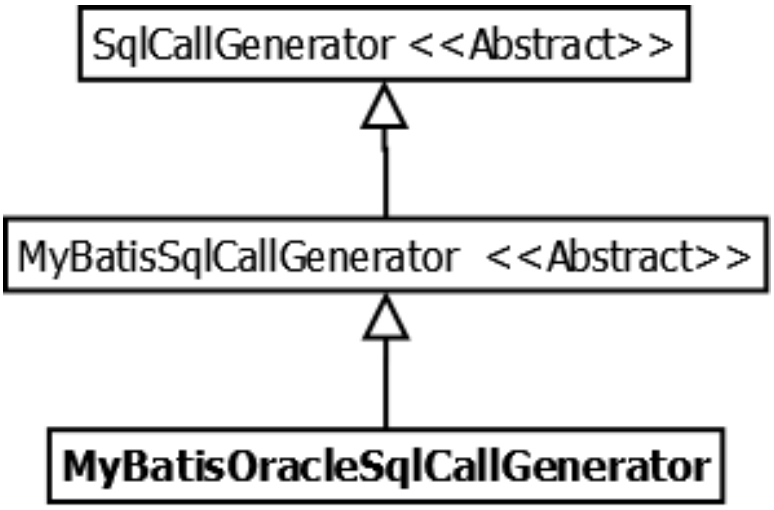


Figura 8. Clases que dan soporte a la generación de las llamadas a procedimientos.

Tras terminar la implementación de estas clases, se tiene la parte necesaria para generar las llamadas a los procedimientos. Más adelante se verá cómo integrar esta parte con el resto. A continuación se va a describir cómo se ha desarrollado la generación del código de los procedimientos.

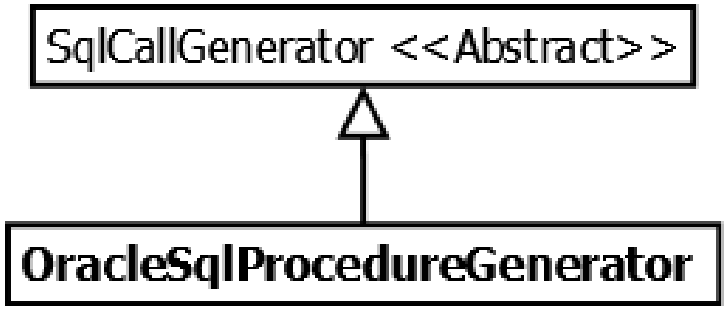


Figura 9. Clases que soportan generación de procedimientos sql.

Para ello se ha definido la clase **OracleSqlProcedureGenerator** que extiende de **SqlCallGenerator**. Esta clase está compuesta por un atributo que implementa la interfaz **QueryGenerator** y que será el encargado de generar el código sql, para Oracle, que ejecuta el procedimiento y que depende de cada operación, es decir, aquel que va entre el **BEGIN** y el **END** del procedimiento. Esta clase también se encarga de crear la estructura de cada procedimiento que posteriormente se situará en el fichero .sql.

El siguiente paso es juntar los componentes que hay hasta ahora para crear el generador de procedimientos. Con ese fin se ha implementado la clase **SqlCallOrQueryGenerator** que debe su nombre a la posibilidad de que un tipo de operación (select, insert, update o delete) sea generado como una llamada a procedimiento o directamente en código sql (como se hacía hasta ahora).

Esta clase extiende de **SqlGenerator** para llevar a cabo la implementación de todas las posibles operaciones, pero utiliza el patrón Composite para poder utilizar el generador de las llamadas a procedimientos, el generador del contenido de los procedimientos y el de las consultas.
A continuación a modo de ejemplo se muestra como se trata una operación de update en esta clase:

```
@Override
public String[] getCustomUpdateQuery(OperationInfo operation) {
   if (useCallForUpdate) {
      String[] procedure = procedureGenerator.getCustomUpdateQuery(operation);
      write(procedure);
      return callGenerator.getCustomUpdateQuery(operation);
   } else {
      return queryGenerator.getCustomUpdateQuery(operation);
   }
}
```

Como se puede observar si se desea generar como procedimiento el flag useCallForUpdate estará a true y se escribirá en el fichero .sql, mediante la llamada al método **private void write(String[] lines)**, definido en la misma clase y encargado de escribir línea a línea el resultado obtenido del generador de procedimientos procedureGenerator. Y además se devuelve el array de String correspondiente a la llamada a dicho procedimiento (obtenido por callGenerator) para que sea generada. En caso de que el flag no sea true se devuelve la consulta generada en sql mediante el queryGenerator.

Por último, se lleva a cabo la implementación de este generador de procedimientos para Oracle 10 utilizando MyBatis, lo cual da lugar a la clase **MyBatisOracle10ProcedureGenerator** que hereda de **SqlCallOrQueryGenerator** y que simplemente se encarga de llamar al constructor de su clase superior con los correspondientes generadores conformando la siguiente jerarquía de clases:

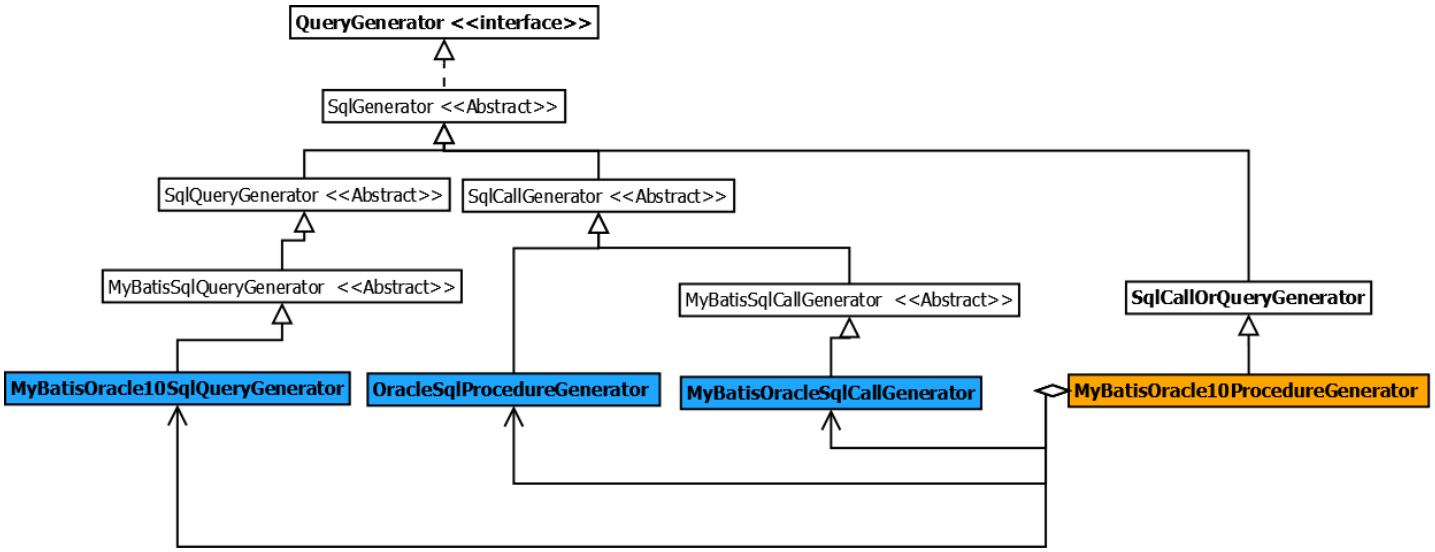


Figura 10. Estructura generación procedimientos almacenados MyBatis-Oracle10.

De esta forma ya sería posible poder agregar el nuevo backend ORACLE_10_PROCEDURE y asociarlo a **MyBatisOracle10ProcedureGenerator.**

Una vez conseguida la generación de procedimientos almacenados se pudo llevar a cabo la generación de paquetes, con el fin de poder agrupar los procedimientos de forma lógica.

En cuanto a las llamadas de procedimientos el cambio que se produce es que ahora no basta con poner el nombre del procedimiento después de la sentencia "call", sino que delante del nombre es necesario colocar el nombre del paquete seguido de un punto:

call nombre_procedimiento ... ;
pasa a ser:
call nombre_paquete.nombre_procedimiento ... ;

La generación de los paquetes en fichero .sql ha requerido modificar la jerarquía de clases, dando lugar a una clase abstracta (**OracleSqlAbstractProcedureGenerator**) que contiene la funcionalidad común de **OracleSqlProcedureGenerator** y la nueva clase encargada de lageneración de la cabecera de los paquetes (**OracleSqlPackageHeaderGenerator**). Además también es necesaria una clase para el cuerpo de los paquetes (**OracleSqlPackageBodyGenerator**) que hereda de **OracleSqlProcedureGenerator.** Quedando de la siguiente manera:

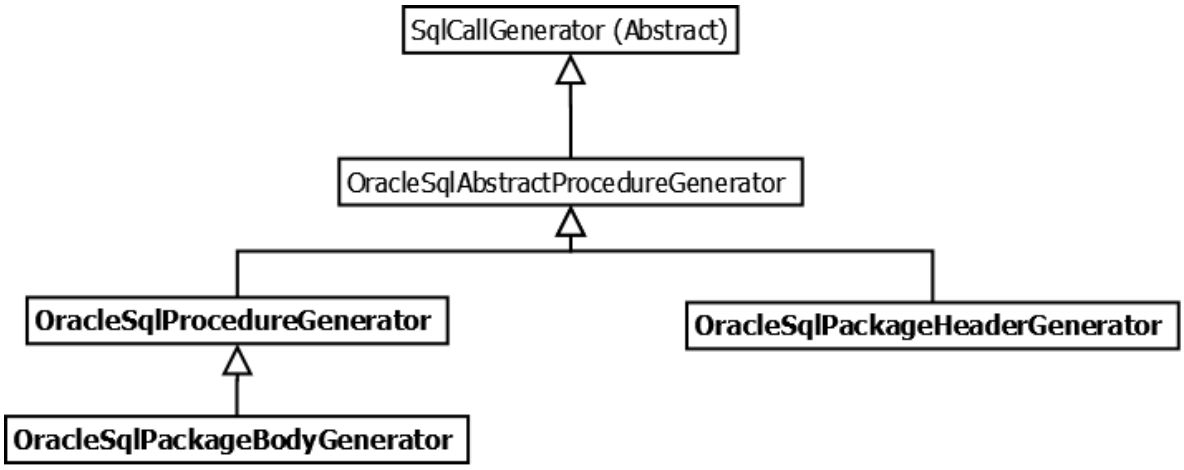


Figura 11. Modificación estructura de clases para la generación de paquetes.

Una vez se dispone de los componentes necesarios para poder llevar a cabo la generación de paquetes es necesario juntarlos, para ello, igual que en el caso de los procedimientos, se ha creado una clase que se componga de todos ellos y se ha denominado **OracleCallPackageOrQueryGenerator**. Tiene una estructura y comportamiento similar a **SqlCallOrQueryGenerator,** salvo que en este caso en vez de un generador de procedimientos, se compone de los generadores de la cabecera y el cuerpo de los paquetes, que son los encargados de escribir en el fichero .sql si se indica que el tipo de la operación debe escribirse como procedimiento, es decir, si está activado el flag que permite seleccionar entre consulta sql o procedimiento.

Para llevar a cabo esta generación de paquetes utilizando Mybatis y Oracle 10 se ha creado la clase **MyBatisOracle10PackageGenerator** que hereda de **OracleCallPackageOrQueryGenerator** y que únicamente llama al constructor de su clase padre de la siguiente forma:

```
public MyBatisOracle10PackageGenerator() {
super( new MyBatisOracleSqlCallGenerator(),
                    new MyBatisOracle10SqlQueryGenerator(),
                    new OracleSqlPackageHeaderGenerator(),
                    new OracleSqlPackageBodyGenerator());
  }
```

Surge así una nueva configuración de backend denominada **ORACLE_10_PACKAGE**.

Implementación de un mecanismo de validación de datos en las entidades generadas.

Motivación.

En muchas ocasiones, a la hora de llevar a cabo el desarrollo de un sistema, se escriben validaciones, que actúan como restricciones sobre los campos, en forma de anotación (por ejemplo Java Bean Validations), lo cual es muy útil por ejemplo

cuando se va a utilizar una base de datos en el sistema, ya que se pueden poner restricciones a los objetos java y pasarlos por un validador antes de enviar la operación a la base de datos y asegurarse de que no va a fallar esa operación al llegar a la base de datos, evitando así una comunicación fallida. También es muy útil para comprobar que la información que procesa el servidor procedente del navegador es consistente con lo previsto, es decir, que cumple con todos los requisitos que luego el sistema da por hecho, mejorando así la seguridad y mantenibilidad del sistema.

Diseño.
El objetivo es permitir añadir anotaciones a los campos de las entidades y operaciones, a modo de validación, de manera que cuando las entidades y las operaciones se generen sus atributos aparezcan con estas restricciones (anotaciones).

En definitiva, lo deseado es que aquellas anotaciones que tengan los campos de entidades y operaciones y no pertenezcan a las anotaciones propias de Uaithne, se mantengan en los campos de las clases generadas.

Por ejemplo, supongamos una entidad de Uaithne, con el fin de contener los datos de un perfil y en la que la edad tiene que estar entre 18 y 150 años y el código postal debe tener 5 dígitos en la parte entera y ningún decimal, definida de la siguiente forma:

```
@Entity
public class_Perfil {
   @Id
   BigInteger id;

   String nombre;

   @Min(18)
   @Max(150)
   int edad;

   String calle;

   @Optional
   @Digits(integer = 5,fraction = 0)
   int codigo_postal;
}
```

En negrita se pueden observar las anotaciones que no pertenecen a Uaithne (@Digits, @Min, @Max) y que son aquellas que deberán aparecer en la clase generada correspondiente a la entidad, quedando la definición de los atributos de esta clase de la siguiente forma:

```
public class Perfil implements Serializable {
    private BigInteger id;
    private String nombre;
   @Min(18)
   @Max(150)
    private int edad;
    private String calle;
   @Digits(integer = 5, fraction = 0)
    private int codigo_postal;

    [...]
}
```

Nota: el generador de código omite el carácter "_" al inicio de un identificador, lo cual se puede utilizar para distinguir fácilmente las clases de entrada para el generador, de las generadas por éste.

Además de este mecanismo para poder añadir anotaciones que aparezcan sobre el código generado, se pensó que podría ser de gran utilidad permitir la generación de anotaciones por defecto sobre un campo de tipo **identificador**, **obligatorio** u **opcional**.

Un claro ejemplo de su utilidad podría ser asignar por defecto la anotación **@NotNull** a todos los campos que sean identificadores y/o obligatorios, es decir, aquellos que no son opcionales, y que por lo tanto no deberían ser nulos.
Este caso implicaría un problema, y es que los tipos de datos primitivos no admiten valores nulos y por tanto no tendría sentido que tuvieran una anotación **@NotNull.** Para hacer más flexible esta generación, los campos identificadores y obligatorios que sean de tipo primitivo no generarán las anotaciones por defecto, a no ser que se indique lo contrario.

Desarrollo.
La primera parte desarrollada, como bien se ha dicho, permite generar en las clases correspondientes de salida, aquellas anotaciones de los campos de entidades y operaciones que se indican en las clases de entrada del generador, y que no pertenecen a Uaithne.

Para ello es necesario, en caso de que los campos tengan anotaciones que no pertenecen a Uaithne:

- Añadir los "imports" que necesiten estas anotaciones. Dado que se pueden tratar tanto de entidades como de operaciones se modifican las clases **OperationTemplate** y **EntityTemplate** de manera que se recorran todos los campos y se vayan añadiendo los imports necesarios de las anotaciones a un HashSet de "imports" que contiene la entidad/operación.

Para obtener las anotaciones de un campo será necesario utilizar reflexión (a nivel de compilador).
Dado que una anotación puede contener más anotaciones internamente es necesario acceder a ellas e incluir los "imports" necesarios de manera recursiva.

- Se modifica la clase **PojoTemplate**, de la cual extienden **OperationTemplate** y **EntityTemplate** ya que ambas tienen campos, para que en el momento en el que se escribe cada campo (en un objeto Appendable), se añadan, después de su documentación (si tuviera) y antes de su declaración, las anotaciones correspondientes, a las que al igual que a la hora de añadir los "imports", se accede mediante reflexión y es necesario el uso de recursión para obtener las que forman parte de otras anotaciones.

  Destacar que todo el material generado por Uaithne, es muy meticuloso con su formato, de manera que parezca que es programado por un ser humano. Por tanto se ha prestado especial atención a la hora de generar las anotaciones, respetando la indentación del texto, tratando adecuadamente la escritura de arreglos, los espacios en blanco, los saltos de línea, etc.

Para la segunda parte del desarrollo, en la cual se posibilita la generación por defecto de anotaciones para campos de tipo **opcional, obligatorio** y/o **identificador**, y si se trata de alguno de los dos últimos y además son tipos de datos primitivos se cancela esa generación a no ser que se especifique lo contrario, se necesita crear unos parámetros de configuración para Uaithne que se añaden a la interfaz de la anotación **@UaithneConfiguration** (utilizada en la clase de configuración de Uaithne) y que son los que se muestran a continuación:

```
Class<?> idValidationAnnotation() default Void.class;
boolean ignoreIdValidationOnPrimitives() default true;
Class<?> mandatoryValidationAnnotation() default Void.class;
boolean ignoreMandatoryValidationOnPrimitives() default true;
Class<?> optionalValidationAnnotation() default Void.class;
```

Los parámetros **idValidationAnnotation, mandatoryValidationAnnotation** y **optionalValidationAnnotation** serán los destinados a indicar que anotaciones se generan en cada tipo de campo, por defecto toman el valor Void.class, lo cual indica que no generan ninguna anotación.

Los parámetros **ignoreIdValidationOnPrimitives** y **ignoreMandatoryValidationOnPrimitives** permiten indicar si los campos que sean de tipos de datos primitivos deben ignorar (true, por defecto) o no (false) la generación de las anotaciones indicadas en **idValidationAnnotation** y **mandatoryValidationAnnotation** respectivamente.

Estos parámetros de configuración son procesados y añadidos como información relevante para llevar a cabo el proceso de generación (en la clase **GenerationInfo**) y dependiendo de ellos se modificará el comportamiento de la clase **PojoTemplate** para añadir las anotaciones que sean necesarias a los distintos tipos de campos.

Aplicación práctica de la solución en el proyecto comercial Unit Linked en desarrollo por parte de la compañía Delonia Software: Incorporación de un sistema de auditoría.

El objetivo es incluir en el proyecto Unit Linked un sistema de auditoría externo que provee distintos métodos para auditar eventos como los accesos y salidas del sistema las modificaciones en usuarios, etc.

El primer paso necesario es obtener una única instancia de la clase que permita registrar las acciones de auditoría, siguiendo el patrón de diseño Singleton.

Tras ello se comienza llevando a cabo la auditoría de accesos (logins) y salidas (logouts) de la aplicación. Para manejar tanto los logins como los logouts se disponen de servlets que controlan estas peticiones. En ese punto es donde se deben auditar estos eventos.

Se han registrado en el sistema de auditoría tanto los accesos y salidas correctos como los incorrectos, indicando datos como la fecha, el resultado (OK/ERROR), el identificador del usuario, la causa del fallo, etc.

El proceso para auditar las modificaciones de usuarios requiere un mayor esfuerzo ya que las operaciones que modifican un usuario de la aplicación Unit Linked son varias y por tanto será necesario introducir interceptores en ellas para poder auditarlas.

Las operaciones que implican cambios en los usuarios son las de creación (o inserción), actualización y borrado.

Se coloca un interceptor antes de la implementación de base de datos de las operaciones de forma que se pueda comprobar si el resultado de esa implementación da algún fallo o no, pudiendo así auditar tanto el caso en el que se realiza la operación de manera correcta como en el que falla.

A continuación se explica con mayor claridad cómo se añade la funcionalidad a la estructura (bus) de la aplicación.

Partiendo de la siguiente representación para Executor (módulo de operaciones) y ExecutorGroup (agrupación o capa):

Figura 13. Executor.

Figura 12. ExecutorGroup

La última capa de la aplicación es la de base de datos (BD) y se implementa mediante un **MappedExecutorGroup** (ofrecido por la arquitectura de Uaithne), que delega la ejecución de la operación al módulo correspondiente, y en caso de que ésta no esté asociada a ninguno devuelve un error

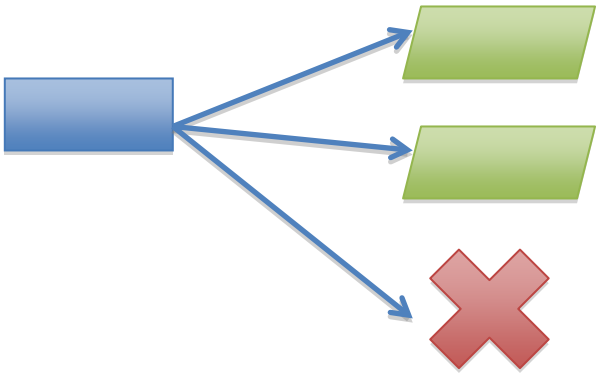


Figura 14. MappedExecutorGroup

En la aplicación, uno de los módulos a los que puede delegar, es el que contiene las operaciones relativas al usuario, que son las que interesan en este caso.

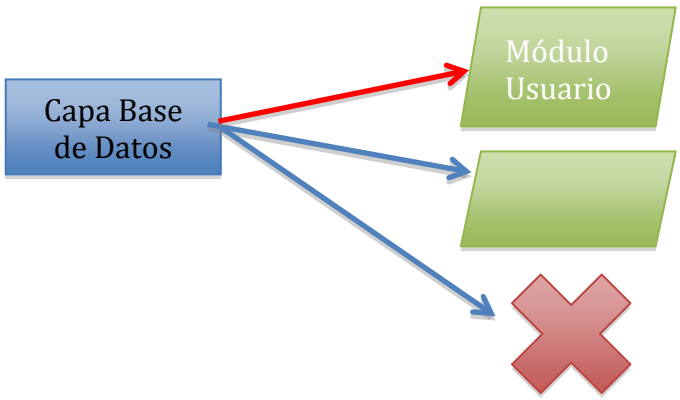


Figura 15. MappedExecutorGroup capa base de datos UnitLinked.

Para poder auditar las operaciones, del módulo usuario, que nos interesan debemos introducir un interceptor en la comunicación marcada en **rojo** de la figura anterior. de modo que quede de la siguiente forma:

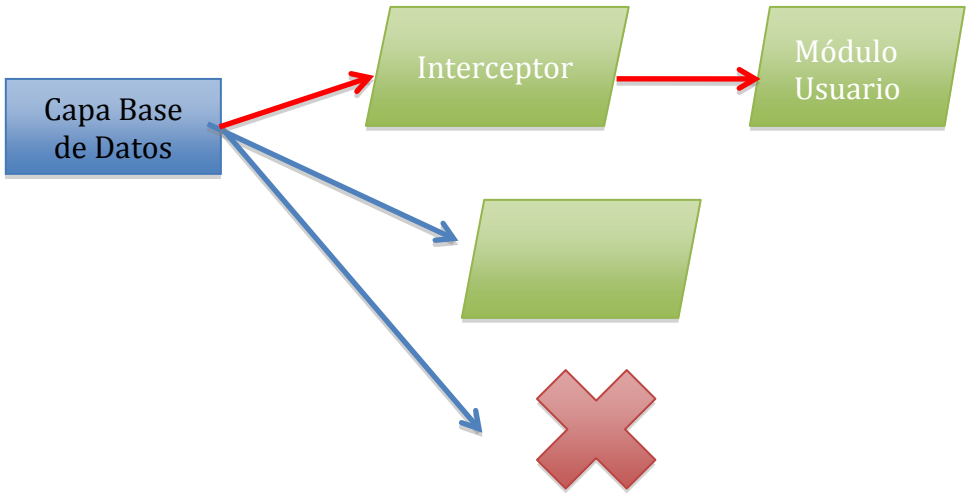


Figura 16. Incorporación interceptor.

Este tipo de interceptor se comporta como un **observador** ya que la operación que recibe es la misma que la que sale de él. La arquitectura que ofrece Uaithne permite definir este interceptor mediante la clase **ChainedExecutor** que actúa como **módulo** (Executor) y delega la ejecución a otro.

De esta manera ya se podría incluir el código necesario, en el interceptor, para implementar la auditoría de las operaciones de modificación de usuarios deseadas. Manteniendo la separación de competencias entre la implementación de base de datos y la implementación de auditoría, consiguiendo que el código se encuentre separado, delegando a cada uno ejecutar el rol que le corresponde.

Conclusiones

Tras concluir la realización del proyecto de fin de grado se puede afirmar que se han logrado los objetivos propuestos.

Por un lado se ha conseguido mejorar el aseguramiento de calidad de la herramienta Uaithne llevando a cabo la implementación de pruebas unitarias sobre su generador , de manera que ahora sea posible evolucionar la herramienta con la confianza de tener una serie de pruebas que aseguren que no se están produciendo comportamientos indeseados al realizar cambios y por tanto aquellos proyectos en los que está ya en uso Uaithne no dejen de funcionar correctamente.
Además se ha elaborado un documento de gran necesidad, ya que en él se ha conseguido sintetizar todo lo necesario para poder utilizar la herramienta Uaithne, facilitando su entendimiento y asimilación mediante numerosos y comunes casos prácticos, que además cuenta con la aprobación de usuarios de la herramienta.

Por otro lado se han conseguido llevar a cabo de manera satisfactoria las tres evoluciones propuestas para la herramienta de Uaithne, y durante su diseño y desarrollo se han adquirido gran cantidad de conocimientos acerca del funcionamiento interno de Uaithne y de cómo realiza su proceso de generación, lo cual será de gran utilidad en caso de seguir participando en la evolución de la herramienta una vez finalizado el trabajo de fin de grado.

Además se han ampliado de manera notable los conocimientos obtenidos durante el transcurso del Grado, que pueden ser de gran utilidad en ámbitos como la programación orientada a objetos (POO), éste es el caso de conceptos como la reflexión y meta-programación, la aplicación de patrones de diseño avanzados, ideas de programación orientada a aspectos, realización de pruebas unitarias utilizando distintos tipos de objetos simulados, etc. También se ha profundizado en campos que rodean el manejo de datos almacenados en bases de datos, debido al trato con los procedimientos almacenados y paquetes, los ORM's, las distintas posibilidades de consulta utilizando MyBatis y Sql, la herencia en bases de datos, etc. Adicionalmente se ha mejorado la capacidad de redacción orientada a documentos de uso.

En definitiva, la realización de este trabajo ha permitido aplicar muchos de los conocimientos adquiridos durante el transcurso del grado, ampliarlos y ponerlos en práctica en un ámbito laboral.

Referencias

[Universidad Tecnológica Nacional, Facultad Regional Buenos Aires]:
Universidad Tecnológica Nacional, Facultad Regional Buenos Aires. (s.f.). *Programación Avanzada con Objetos Introducción a la metaprogramación.* Recuperado el Julio de 2014, de https://sites.google.com/site/programacionhm/conceptos/metaprogramacion/intro

[Bernard]:
Bernard, E. (s.f.). *Bean Validation specification 1.1 Final.* Recuperado el Julio de 2014, de http://beanvalidation.org/1.1/spec/#introduction

[Bolaños Alonso,2008]:
Bolaños Alonso, D. (2008). *Pruebas de software y JUnit.* Pearson Prentice Hall.

[Gamma, Helm, Johnson, & Vlissides, 1994]:
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns, Elements of Reusable Object-Oriented Software.*

[Escuela Politécnica Superior UAM, 2013]:
Ingeniería del Software, Unidad 5 Codificación Pruebas y Finalización2013

[JUnit]:
JUnit. (s.f.). Recuperado el Julio de 2014, de http://junit.org/

[Keith & Schincariol, 2009]:
Keith, M., & Schincariol, M. (2009). *Pro JPA 2: Mastering the Java™ Persistence API.*

[Kuaté, Bauer, King, & Harris, 2009]:
Kuaté, P. H., Bauer, C., King, G., & Harris, T. (2009). *NHibernate in Action.*

[López Rodriguez]:
López Rodriguez, J. R. (s.f.). Object/Relational Mapping, licencia creative commons Reconocimiento-NoComercial-CompartirIgual 3.0 http://creativecommons.org/licenses/by-nc-sa/3.0/deed.es_ES.

[Microsoft]:
Microsoft Developer Network, Procedimientos almacenados (motor de base de datos)

[MyBatis]:
MyBatis. (s.f.). Recuperado el Julio de 2014, de http://mybatis.github.io/mybatis-3/es/

[Paz Rojas, 2007]:
Paz Rojas, J. L. (Septiembre de 2007). Programación por Chequeo. *Trabajo especial de grado* . Venezuela.

[Escuela Politécnia Superior UAM, 2012]:
Proyecto de Análisis y Diseño de Software, Tema 3 Pruebas unitarias con JUnit2012

[Sun Microsystems]:
Sun Microsystems. (s.f.). *JavaBeans™ Specification 1.01 Final Release.* Recuperado el Julio de 2014, de http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/

[Tanter, 2002]:
Tanter, E. (Noviembre de 2002). *Reflexión, metaprogramación y programación por aspectos DCC–University of Chile- Ecole des Mines de Nantes (France).* Recuperado el Julio de 2014, de http://users.dcc.uchile.cl/~etanter/courses/tanter-jcc02.pdf

Anexo A

Introducción práctica a Uaithne

Josué Fernández León

JULIO 2014

Introducción

A continuación se presentan una serie de conocimientos necesarios para poder utilizar Uaithne, explicando aquellos conceptos más importantes acompañados de casos prácticos muy comunes, que permitan al lector situar estos conceptos dentro de un contexto.

Entity

Es una representación de una entidad de datos (tabla), que por defecto, genera operaciones de inserción, borrado, actualización y selección por id, a menos que se hayan desactivado en la configuración. Una clase se marca como entidad con la anotación "@Entity", pero además ésta se debe encontrar dentro de una clase marcada como módulo de operaciones (**@OperationModule**) para que se puedan generar las operaciones por defecto. Un módulo de operaciones actúa como una agrupación lógica de operaciones que permite luego tratarlas como un único conjunto.

Ejemplo: se supone una tabla que tiene el fin de almacenar vuelos de una base de datos de una compañía aérea.

```
@OperationModule
publicclass_vuelos {

        @Entity
        @Doc({"Tabla de vuelos de la bd"})
        class_Vuelo {
          @Id
          BigIntegerid;

          @MappedName("cliente")
          BigIntegerpasajero;

          BigIntegercodigoOrigen;

          BigIntegercodigoDestino;

          Date fechaSalida;
        }
}
```

Como se puede observar, se ha creado una clase **_vuelos** marcada como un módulo de operaciones. Destacar que dado que el nombre de un módulo se transforma en un paquete con las operaciones, es conveniente que esté en plural y empiece por minúscula. También merece la pena resaltar que el generador de código omite el

carácter "_" al inicio de un identificador, lo cual se puede utilizar para distinguir fácilmente las clases de entrada para el generador, de las generadas por éste.

Dentro de esta clase se encuentra la clase _Vuelo que está marcada como una entidad y además dispone de una anotación "**@Doc**" que permite añadir documentación que posteriormente se generará junto con la clase. Esta anotación recibe un **arreglo de string**, donde cada ítem será generado como una línea distinta de la documentación.

La clase contiene los distintos campos de la tabla "vuelo" de la base de datos. Destacar que la anotación "**@Id**" permite marcar un campo como identificador, lo cual es obligatorio para que un Entity pueda generar las operaciones mencionadas más adelante; aunque el id, por defecto, se autogenera, se puede evitar de la siguiente manera **@Id(autogenerated= false)**. Si se desea profundizar más acerca de las posibilidades de configuración del Id se puede consultar la sección destinada a ello (ver: `Configuración avanzada de Id`).Además mediante "**@MappedName**" se puede indicar el nombre de la columna, o incluso de la tabla (si se aplica sobre una entidad Entity o EntityView), en la base de datos, si éste es distinto del nombre que tiene el campo o la clase.

Clases Generadas.

Tras compilar el proyecto en el paquete "model" de las clases generadas se encontrará la clase Vuelo.java que contendrá los atributos que se han especificado, así como los **getters** y **setters** de éstos, el método **toString** y los métodos **equals** y **hashCode**; estos dos últimos al haber especificado un id (mediante **@Id**) se implementarán en función a dicho campo.
Además de dos constructores, uno con todos los atributos, incluido el id y otro sin éste último. Además destacar, que en caso de que hubiese atributos opcionales (@Optional) éstos no aparecerían en el constructor.

Además en el paquete "operations" se genera el paquete vuelo con las clases de la Entity que representan las operaciones de **inserción, borrado por id, selección por id** y **actualización,** incluyendo **update** y **merge** (update de aquellos campos distintos de null), y las clases generadas relativas al módulo de operaciones:

- **VuelosExecutor:** interfaz para la implementación de la funcionalidad provista por el módulo, es decir, en el momento que se desee implementar la funcionalidad de las operaciones, la clase destinada a ello deberá implementar esta interfaz.
- **VuelosAbstractExecutor:** Esta clase abstracta generada permitiría realizar la implementación de las operaciones, directamente extendiendo esta clase, que ya implementa la interfaz VueloExecutor, y provee la implementación de los elementos básicos comunes.

- Además se generan dos clases a modo de interceptores[3], que permiten añadir lógica adicional y delegar la ejecución de las operaciones a otra implementación del módulo (**VuelosChainedExecutor)** o en un grupo de módulos (**VuelosChainedGroupingExecutor**). Para ello se crearía una clase que extendería de alguna de las mencionadas y mediante la sobrescritura de métodos se podría añadir la lógica deseada y delegar la ejecución a la implementación original mediante "super".
  Dado que el concepto de interceptor, de primeras, puede ser difícil de entender , se puede comprender con mayor claridad si se lee el apartado de `Configuración de comportamientos comunes` llamado `Intercepta`.

EntityView

Es similar a una Entity, salvo porque no genera ninguna operación por defecto. Se suele utilizar para representar consultas que obtienen un subconjunto de datos, como por ejemplo algunas columnas de una o varias tablas. Para ello se marca una clase con la anotación **@EntityView.**

@MyBatisMapper

Si a un módulo de operaciones (**@OperationModule)** se le añade esta anotación, se generará la lógica de acceso a base de datos usando MyBatis. Excepto para aquellas operaciones que estén marcadas con la anotación **@Manually**, que indica que la implementación se realizará manualmente.

NOTA: la anotación **@Manually** aplicada sobre un campo indica que ese campo va a ser manejado (implementado) manualmente por el programador, por lo que se ignora de toda query.

Siguiendo el ejemplo anterior de la entidad de vuelos, al añadir la anotación al módulo y compilar se puede comprobar que en los generados, dentro del paquete myBatis, se han generado los ficheros **MyBatisVueloMapper.java** y **MyBatisVueloMapper.xml**.

La clase MyBatisVueloMapper (de MyBatisVueloMapper.java) implementa la interfaz **VueloExecutor,** ya que el objetivo de esta clase es realizar la implementación de acceso a base de datos utilizando MyBatis, del módulo. Y tiene un atributo que corresponde con el proveedor de sesión Sql, para poder establecer conexión con la base de datos.

[3]Un interceptor es un elemento que se interpone en el flujo de ejecución de la operación permitiendo modificar su comportamiento; por ejemplo, se puede decir que un trigger de base de datos actúa como un interceptor de las operaciones que se realizan sobre una tabla.

El archivo MyBatisVueloMapper.xml contiene las operaciones de bases de datos a nivel de acceso a tablas escritas en XML de MyBatis.

Es importante no olvidar añadir **MyBatisVueloMapper.xml** a la configuración de MyBatis y **MyBatisVueloMapper.java** a la `Configuración del bus`.

Operaciones

Hay distintas anotaciones para indicar que una clase perteneciente a un (@OperationModule) es una operación. Pero todas ellas admiten una serie de parámetros. A continuación se procederá a explicar esos posibles parámetros y posteriormente se indicará cuales pueden ser especificados en cada una de las operaciones.

Parámetros:

- **result :** indica el tipo resultado de la operación.
- **limit :** indica si solo se debe devolver el primer registro de la operación. Por defecto: **false**.
- **related:** si el resultado de la operación no es una clase marcada con la anotación @Entity o @EntityView se usa el Entity o EntityView indicado en este parámetro para obtener la información para generar la query; el valor por defecto es **Void** lo que provoca que no se tenga en cuenta.
- **distinct**: permite indicar si la operación debe traer resultados distintos, no repetidos; el valor por defecto es **false**.
- **returnLastInsertedId**: permite indicar si el resultado es el id del registro insertado o si su valor es false el resultado es el número de registro afectados; el valor por defecto es **true**.
- **value**: se utiliza en las operaciones que actúan sobre una entidad, es decir, la reciben y operan sobre ella. Este parámetro indica dicha entidad (Entity o EntityView).

	Posibles parámetros					
Anotación de la operación / uso	result	limit	related	distinct	returnLast InsertedId	value
@Operation / operación básica sin comportamiento especial. Lo definirá el programador por completo.	☑					
@SelectOne/ trae 1 registro	☑	☑	☑			
@SelectMany/ varios	☑		☑	☑		
@SelectPage/ página	☑		☑	☑		
@SelectCount	☑		☑			
@Insert			☑		☑	

@Update			☑			
@Delete			☑			
@SelectEntityById	☑	☑				
@InsertEntity					☑	☑
@UpdateEntity						☑
@DeleteEntityById			☑			
@SaveEntity					☑	☑
@MergeEntity						☑

La anotación "**@SetValue**" aplicada sobre un campo de una operación de actualización, indica que es un campo a ser actualizado, y el resto serán aquellos que forman parte de la condición que se debe cumplir para actualizar.

Casos prácticos de uso de **operaciones** y **EntityViews**:

1. **Suponer, el caso más sencillo en el que se desea obtener un vuelo de una tabla de vuelos dado su identificador.**

```
@SelectOne(result  = _Vuelo.class)
static class _SelectVueloFromId {
    @Id
     BigInteger id;
}
```

Se ha utilizado una operación de tipo **@SelectOne** que tiene como resultado la entidad Vuelo, para ello se ha marcado una clase con esa anotación. Los campos de la clase se transformarán, tras la generación, en condiciones del "where" de la consulta separadas por la condición AND.

Comentar que existe la posibilidad de tener campos opcionales, es decir, que se añaden a la condición del where si tienen un valor no nulo, para ello se utiliza la anotación @Optional sobre el campo. Es posible ver un ejemplo de uso de esta anotación en el caso práctico 5.

2. **Imaginar que se desea una operación sobre la entidad de vuelos, que dado un id nos devuelva el id del pasajero (misma tabla), para ello se propone la siguiente solución:**

```
@SelectOne(result = BigInteger.class, related = _Vuelo.class)
@CustomSqlQuery(select = "cliente")
static class _SelectPasajeroFromIdVuelo {
        BigInteger id;
}
```

Como se puede observar se ha utilizado una operación de tipo **@SelectOne**, debido a que retornará un único resultado, en este caso de tipo BigInteger, y que tiene como "**related**" la Entity _Vuelo, de la que obtiene la información para generar la query.
La anotación **@CustomSqlQuery** permite personalizar las distintas partes de una query como pueden ser el "from", "where", "beforeGroupByExpression", "afterQuery", etc. En este caso se ha especificado el "**select**", para indicar que el campo que se ha de retornar es el que se denomina "cliente".
Se especifica la clase con el nombre que queramos darle a la operación y se declara dentro de ella la variable correspondiente al id, que formará parte de la condición que aparezca en el "where".

3. **Ahora supongamos que deseamos obtener el nombre del pasajero dado su id de una tabla "Pasajero". Debido a la sencillez de la consulta se opta por la siguiente solución:**

```
@SelectOne(result = String.class)
@CustomSqlQuery(query = "select nombre from pasajero where id = {{id:value}}")
static class _SelectNombreFromIdPasajero {
   BigInteger id;
}
```

En este caso se ha indicado la query manualmente, y para indicar el parámetro que se va a tomar de la clase se especifica mediante {{nombre del campo : regla con la que procesarlo}}.
Todo aquello que va entre "{{}}" se procesará, una vez se haya procesado el resto de la query. Poner {{id:value}} habría sido equivalente a indicar directamente mediante MyBatis (ya que es el framework utilizado en este caso) "#{id, jdbcType=NUMERIC}".
Para conocer más en detalle las posibles reglas que se pueden incluir en la query para su traducción, en el momento final de procesamiento de la misma, consultar el apartado `Referencia a campos dentro de una query`.

4. **Se da la situación en la que se quiere, a partir del código de origen y destino de un vuelo, obtener todos los pasajeros que realizarán ese trayecto y la fecha del mismo, con la condición de que la fecha sea mayor que la actual y ordenado por este mismo campo de manera ascendente.**

En primer lugar se crea un "EntityView" que representa el subconjunto de datos que contiene un pasajero y una fecha, y que se utilizará para alojar el resultado de la operación.

```
@EntityView
@MappedName("vuelo")
static class _PasajeroYFecha {
```

```
    @MappedName("cliente")
    BigIntegerpasajeroAvion;
    @MappedName("fechaSalida")
     Date fechaSalidaAvion;
}
```

La operación que se necesita en este caso es un **@SelectMany** (aunque también podría tratarse de un **@SelectPage** si se desean obtener los resultados paginados), dado que la consulta podrá retornar varios resultados.
En este caso se especifica como resultado la nueva EntityView y mediante @CustomSqlQuery se indica que el **orderBy** se hará mediante "fechaSalida", además con **afterWhereExpression** se indica la condición de que la fecha sea mayor que la actual, que se añadirá literalmente a la consulta después de la expresión "where".

```
@SelectMany(result = _PasajeroYFecha.class)
@CustomSqlQuery(orderBy = "fechaSalida",afterWhereExpression = "and fechaSalida&gt; current_timestamp")
static class _SelectPasajeroYFechaFromOrigenYDestino {
    BigIntegercodigoOrigen;
    BigIntegercodigoDestino;
}
```

Como se puede observar en lugar del símbolo '>' aparece la expresión '>', esto se debe a que es la notación que se usa en XML para el símbolo de mayor. Los símbolos que deben "escaparse" en XML de manera especial son:

Símbolo	XML
"	"
'	'
<	<
>	>
&	&

5. **Suponer que se quiere hacer la misma funcionalidad que la operación anterior, pero en este caso se desea poder introducir la fecha con la que comparar por parámetro, y en caso de que no se introduzca omitir la condición. Este caso es muy útil para elaborar filtros.**

```
@SelectMany(result = _PasajeroYFecha.class)
@CustomSqlQuery(orderBy = "fechaSalida")
static class _SelectPasajeroYFechaFromOrigenYDestino2 {
    BigInteger codigoOrigen;
    BigInteger codigoDestino;
```

```
    @Optional
    @UseComparator(Comparator.LARGER)
    Date fechaSalida;
}
```

Con la anotación **@Optional** se indica que el campo se incluya en la operación sólo si no es **nulo**. **@UseComparator** permite indicar el **criterio de comparación** entre la columna de la base de datos y el valor del campo. Los posibles valores se encuentran en la enumeración **Comparator** y por defecto el comparador utilizado es **EQUAL,** a menos que se trate de una lista, en cuyo caso el comparador será **IN**.
Si el comparador no se encuentra en la enumeración, se puede especificar manualmente utilizando la anotación **@UseCustomComparator,** si se desea conocer más acerca del uso de esta anotación consultar el apartado `Comparadores especificados manualmente.`

Si se quisiera poder **indicar por parámetro** el/los campos por los que ordenar la consulta para ello se ha de utilizar la anotación "**@OrderBy"** aplicada sobre un campo de tipo **String**, y no especificar el orderBy como parámetro de la anotación @CustomSqlQuery.

```
@OrderBy
String orderBy;
```

De esta manera cuando se establezcan los parámetros de la operación, se podrá establecer el valor de este campo con uno o varios nombres de propiedades de la clase resultado, separados por comas, y opcionalmente seguidos de "desc" o "asc".
Ejemplo: "fechaSalidaAvion desc, pasajeroAvion"

NOTA: Es muy común el deseo de que en caso de que no se especifique el campo de ordenación, la consulta muestre los resultados en un determinado orden por defecto. Para ello se suele utilizar la anotación **@DefaultValue** sobre el campo marcado con **@OrderBy.**
Ejemplo: suponer que tenemos el campo OrderBy y queremos que cuando no se especifique, la consulta aparezca ordenada por el campo "pasajeroAvion" .

```
@OrderBy
@DefaultValue("pasajeroAvion")
String orderBy;
```

La anotación **@DefaultValue** recibe un String pero puede aplicarse sobre cualquier tipo de campo, si este campo es booleano, carácter, numérico (incluyendo BigDecimal y BigInteger) o String se realiza la conversión al tipo,

en cualquier otro caso, el String, se interpretará como el código java requerido para inicializar el campo.
La anotación **@DefaultValue** no tiene que estar asociada a un campo marcado como **@OrderBy,** aunque el ejemplo que se ha mostrado describe su caso de uso más común.

6. **En este caso se desea poder modificar la fecha de salida del vuelo dado su id.**

```
@Update(related = _Vuelo.class)
static class _ModificarFechaVuelo  {
BigIntegerid;
        @SetValue
        Date fechaSalida;
}
```

Para ello en se creará una operación de "**@Update"** que tendrá marcado con la anotación "**@SetValue"**, aquellos campos que se desean modificar, y el resto seguirán formando parte de la condición, como lo hacen habitualmente.

7. **Para aquellas ocasiones en las que se necesita hacer una tarea compleja que forma parte de la funcionalidad del sistema, sin ser algo específico de una capa intermedia de la aplicación, por ejemplo, en este caso la tarea de elaborar un billete de vuelo en PDF, que pueda necesitar consultar la BD, utilizar alguna plantilla de billete… , se puede utilizar la anotación "@Operation" que permitirá una implementación especificada manualmente por el programador, abstrayéndose de la comunicación entre capas.**

```
@EntityView
static class _Billete {
   StringnombreDocumento;
   byte[] contenido;
}

@Operation(result = _Billete.class)
static class _GenerarBillete {
   BigIntegeridVuelo;
}
```

Se ha creado una operación que devuelve un EntityView llamado billete que contendrá el nombre del documento y el contenido del billete una vez elaborado. La operación recibe el id del vuelo en cuestión y no se generará ninguna implementación, el

programador deberá especificarla manualmente, por ejemplo, en una clase que extienda de **VueloAbstractExecutor** y haciendo **@Override** del método **public Billete generarBillete(GenerarBilleteoperation);.**

Operaciones de consulta complejas

En muchas ocasiones las operaciones de consulta aumentan en complejidad, y extensión notablemente, por ello existe la posibilidad de especificar la consulta en un mapper xml directamente para **MyBatis.** Para ello se debe emplear la anotación **@MyBatisCustomSqlStatementId,** indicando como "value" un String formado por el **namespace_del_mapper.id_consulta**.

En caso de tratarse de una operación de **SelectPage,** mediante el parámetro "**countStatementId**" se obtendrá la cantidad de elementos en el resultado de la operación. Para el resto de las operaciones este parámetro es ignorado y su valor por defecto es un string vacío.

Es **importante** no olvidar añadir el mapper a la **configuración de MyBatis**.

Consejo: Generar la operación sin la anotación **@MyBatisCustomSqlStatementId** para obtener una primera aproximación de cómo incluirla en nuestro nuevo mapper, observando los tipos parámetro, resultado, viendo como se especifican los valores a ser incrustados en la query, etc.

Operaciones centradas en entities (Entity o EntityView)

Existe la posibilidad de especificar una serie de operaciones sobre un Entity o EntityView que asuman los campos de esa entidad para formar la consulta, son:

@SelectEntityById : genera una operación que devuelve el resultado de un select por id. Tiene como parámetros **value** que se corresponderá con la clase de la entidad en cuestión y **limit.** booleano que en caso de tomar el valor true hará que la operación retorne el primer resultado de la consulta. Si por alguna razón la consulta retorna más de un resultado y no se limita a tomar el primero se provoca una excepción.

Ejemplo:

```
@EntityView(related= _Vuelo.class)
public class _VueloCliente {
   BigInteger id;
   BigInteger pasajero;
 }

@SelectEntityById(result = _VueloCliente.class, limit = true )
```

```
static class _selectVueloClienteById {}
```

Así se obtendría el id y el pasajero del registro, que son los campos indicados en la entidad **_VueloCliente,** de la tabla de vuelos, ya que la entidad tiene un related a _Vuelo, lo cual hace que tome el nombre de la tabla de la entidad indicada en este campo (si VueloCliente tuviese una anotación @MappedName sobre la entidad prevalecería ese nombre a pesar del related) y además permite la herencia lógica de los campos de una entidad a otra, es decir, los campos de la entidad **VueloCliente** que se denominen igual que los de **Vuelo** heredarán sus características (anotaciones de Uaithne), para comportarse de la misma manera.

@InsertEntity: genera una operación de insert con los campos de la entidad y devuelve el id del registro insertado, a no ser que se le dé el valor de **false** al parámetro **returnLastInsertedId**, en cuyo caso retornaría el número de registros afectados (en este caso uno)**.**

Ejemplo:

```
@InsertEntity(_VueloCliente.class)
static class _insertVueloCliente{}
```

Utilizando MyBatis con ORACLE_10 genera:

```
<insert id='insertVueloCliente'
parameterType='org.uaithne.samples.vuelo.model.VueloCliente'
useGeneratedKeys='true' keyProperty='id' keyColumn='id'>
insert into
   mappedVuelo
(
   id,
   cliente
 ) values (
   mappedVuelo_id_seq.nextval,
   #{pasajero, jdbcType=NUMERIC}
)
</insert>
```

Para especificar los datos de la inserción a la operación generada se debe establecer la propiedad **value**, del mismo tipo que la entidad.
En el caso del ejemplo será de tipo **VueloCliente**.

@UpdateEntity: genera una operación de update de los campos de la entidad, a excepción del id que formará parte del where, con el fin de saber que registro actualizar.

Ejemplo:

```
@UpdateEntity(_VueloCliente.class)
 static class  _updateVueloCliente {}
```

Para especificar los datos de la actualización a la operación generada se debe establecer la propiedad **value**, del mismo tipo que la entidad.

@DeleteEntityById: genera una operación que permite eliminar registros por id de la tabla asociada a la entidad. Esta entidad se indica a través del parámetro **related.**

Ejemplo:

```
@DeleteEntityById(related = _VueloCliente.class)
static class  _deleteVueloClienteyById {}
```

La operación generada permite indicar el id del registro a eliminar en la propiedad **id** de la misma.

@MergeEntity: genera una operación de mezcla, es decir, actualiza los campos de la entidad, a excepción de aquellos que son null y del id que formará parte del where, con el fin de saber que registro actualizar.

Ejemplo:

```
@MergeEntity(_VueloCliente.class)
static class  _mergeVueloCliente {}
```

Para especificar los datos de la actualización a la operación generada se debe establecer la propiedad **value**, del mismo tipo que la entidad.

@SaveEntity: genera una operación que equivale a una operación de insert si el id de la entidad es null, y a una operación de update en caso contrario. La operación devuelve el id del registro insertado o actualizado, a no ser que se le dé el valor de **false** al parámetro **returnLastInsertedId**, en cuyo caso retornaría el número de registros afectados (en este caso uno).

Ejemplo:

```
@SaveEntity(_VueloCliente.class)
 static class  _saveVueloCliente {}
```

Para especificar los datos del "save" la operación generada se debe establecer la propiedad **value**, del mismo tipo que la entidad.

Comparadores especificados manualmente

En caso de que la anotación **@UseComparator** no disponga del criterio de comparación que se desea, se puede especificar manualmente este criterio mediante la anotación **@UseCustomComparator,** indicando como "**value**" un String que contendrá la implementación de dicho comparador.

NOTA: Cuando se especifica únicamente el parámetro value de una anotación en vez de "**value = valor_que_se_desee"** se puede poner directamente "**valor_que_se_desee"**. También es el caso de @Doc, @MappedName, etc. Ejemplo: **@UseComparator(Comparator.**LARGER**)** es equivalente a **@UseComparator(value = Comparator.** LARGER**).**

Para hacer referencia al campo en cuestión dentro de este String se pueden utilizar las siguientes expresiones:

Expresión	Tras procesado se convierte en:
[[column]]	El nombre de la columna de la tabla
[[name]]	El nombre del campo
[[value]]	El valor del campo
[[jdbcType]]	El tipo jdbc del campo

Ejemplo:
Imaginar que se quieren obtener los vuelos de un pasajero, pero sólo se conoce su DNI. Para ello se necesita consultar la tabla "pasajero" y sacar el id asociado con ese DNI. (El DNI no se corresponde con el id).
Una posible solución sería:

```
@SelectMany(result = _Vuelo.class)
static class _SelectVueloFromDNIPasajero {
        @UseCustomComparator("cliente in (select id frompasajero wheredni =
        [[value]])")
        String dniPasajero;
}
```

NOTA: Existe un **CASO ESPECIAL** en el que es posible usar las dos anotaciones (**@UseComparator y @UseCustomComparator**) a la vez. Se utiliza cuando se desea añadir código a una comparación predefinida.

Ejemplo:
Supongamos un campo que se tiene un campo con la anotación **@UseComparator(Comparator.SMALLER)** y antes o después de esa comparación se desea añadir código.

Para ello se añade al campo la siguiente anotación:
@UseCustomComparator("código deseado... [[condition]] código deseado...")

[[condition]]se remplazará por el comparador que tenga el campo por defecto o, en caso de existir, el especificado en la notación **@UseComparator.**

Referencia a campos dentro de una query

Dentro de las consultas se pueden referenciar campos mediante dobles llaves "{{ }}" en las que se podrán incluir sentencias que sigan las estructuras descritas a continuación, para ser traducidas e incorporarse de ésta manera a la query.

- **{{campo}}->Comparación por defecto**
  Se convierte en la comparación por defecto (EQUAL, IN o especificado mediante la anotación **@UseComparator**) entre la columna correspondiente al campo y su valor.

  Ejemplo:
  Supongamos la siguiente operación similar a la del ejemplo de operación 3:

  ```
  @SelectOne(result = String.class)
  @CustomSqlQuery(query = "select nombre from pasajero where {{idPasajero}}")
  static class _SelectNombreFromIdPasajero {
     @MappedName("id")
     BigInteger idPasajero;
  }
  ```

  Utilizando como framework MyBatis la traducción resultante sería:
  `select nombre from pasajero where id = #{idPasajero,jdbcType=NUMERIC}`

  NOTA: En caso de que el campo tuviera la anotación **@UseCustomComparator** se sustituiría directamente por el valor de ésta (sin comparar el valor de la columna y el campo).

- **{{campo:comparador o regla }}**
  Se puede distinguir entre una serie de reglas definidas y comparadores (o condiciones) en versión de signo o texto.

Las reglas definidas son:

Expresión	Se convierte en:	Traducción ejemplo MyBatis anterior
{{campo:value}}	El valor del campo	**#{idPasajero,jdbcType=NUMERIC}**
{{campo:name}}	El nombre del campo	**idPasajero**
{{id:column}}	La columna de la tabla	**id**
{{id:jdbcType}}	El tipo jdbc del campo	**,jdbcType=NUMERIC**
{{id:typeHandler}}	El typeHandler del campo en caso de disponer.	En este ejemplo no aparecería nada porque el campo no tiene especificada la anotación **@MyBatisTypeHandler()**que lleva como parámetro una clase que debería implementar la interfaz TypeHandler de MyBatis. Si por ejemplo se especificase IdPasajeroTypeHandler.class, se transformaría en : **,typeHandler=nombre_del_paquete.IdPasajeroTypeHandler**

NOTA: En MyBatis **{{campo:value}}** se transforma en **#{[[name]][[jdbcType]][[typeHandler]]}**

Los comparadores definidos son:

Comparadores (Equivalentes separados por comas)
=, ==, equal
!=, <>, notEqual
=?, ==?, equalNullable
=!?,==!?, equalNotNullable
!=?,<>?, notEqualNullable
!=!?,<>!?, notEqualNotNullable
<, smaller
>, larger
<=, smallAs
>=, largerAs
in

notIn
like
notLike
ilike, likeInsensitive
notIlike, notLikeInsensitive
startWith
notStartWith
endWith
notEndWith
istartWith, startWithInsensitive
notIstartWith, notStartWithInsensitive
iendWith, endWithInsensitive
notIendWith, notEndWithInsensitive
contains
notContains
icontains, containsInsensitive
notIcontains, notContainsInsensitive

La expresión {{campo:**comparador**}}, se traducirá en una comparación, entre la columna de la tabla y el valor del campo, dependiendo del **comparador** indicado.
En el caso de los valores en versión de texto, tanto para reglas definidas como comparadores, serán aceptados también con todos los caracteres en mayúsculas y separando las palabras mediante guiones bajos. Ejemplo: **notEndWithInsensitive** quedaría como **NOT_END_WITH_INSENSITIVE**.

Un caso especial es {{campo:**condition**}}, que se comporta exactamente igual que {{campo}}.

- **{{campo:custom:condición especificada por el usuario}}**
  Existe la posibilidad de especificar el comparador manualmente de forma equivalente al uso de la anotación **@UseCustomComparator**. Indicando como tercer parámetro la cadena que contendrá el código de dicho comparador.

  Ejemplo: teniendo el mismo caso que en el ejemplo de los `Comparadores especificados manualmente`, en el que se deseaba obtener los vuelos de un pasajero a partir de su DNI.

```
@SelectMany(result = _Vuelo.class)
```

```
static class _SelectVueloFromDNIPasajero {
        @UseCustomComparator("cliente in (select id frompasajero
wheredni = [[value]])")
        String dniPasajero;
}
```

Podemos obtener el mismo resultado de la siguiente forma:

```
@SelectMany(result = _Vuelo.class)
@CustomSqlQuery(where = "{{dniPasajero:custom:cliente in (select id
from pasajero wheredni = [[value]])}}")
static class _SelectVueloFromDNIPasajero2 {
  String dniPasajero;
}
```

Como se puede observar se ha añadido a la parte del "where" de la consulta un comparador para el campo dniPasajero, especificado manualmente y cuyo valor será la cadena"cliente in (select id from pasajero wheredni = [[value]])", donde "value" acabará siendo remplazado por el valor del campo, al igual que en el ejemplo original.

En caso de utilizar esta estructura sin el último parámetro **{{campo:custom}}** se utilizará el comparador por defecto.

- Si se desea poder ejecutar código en el caso de que un campo sea null o no, se dispone de las siguientes sentencias:

  {{campo:ifNull:separador opcional}}
  Comienza un bloque de código que se ejecutará si el campo especificado es nulo. Además se puede añadir un separador de manera opcional que se colocará tras la comprobación de si el campo es nulo, por ejemplo, se puede usar este campo para concatenar más condiciones mediante un or, and o cualquier otro separador.

  {{campo:ifNotNull:separador opcional}}
  El mecanismo es idéntico al caso anterior, sólo que esta vez la condición que se debe cumplir, es que el campo no sea nulo.

  {{campo:endIf }}
  Permite finalizar cualquiera de los dos bloques iniciados mediante las sentencias anteriores.

  Tanto **ifNull, ifNotNull** como **endIf,** también serán aceptados si se escriben en mayúsculas, separando las palabras mediante guiones bajos. Ejemplo: **ifNotNull** quedaría como **IF_NOT_NULL.**

Configuración avanzada de Id

Como ya se ha mencionado se ha de marcar un campo de una clase como Id mediante (**@Id**), para poder llevar a cabo las operaciones de inserción, actualización y borrado correctamente.

Por defecto este Id se autogenera bien mediante un auto-incremental o una secuencia, dependiendo de las posibilidades que ofrezca el DBMS (Database Management System). Por ejemplo, Oracle no maneja campos auto-incrementales, por lo tanto se gestiona el uso del Id mediante una secuencia .
Relacionado con este tema, en caso de que se utilice el framework de MyBatis, a la hora de definir la configuración de Uaithne (**@UaithneConfiguration**), se puede modificar la configuración de MyBatis y aparte de indicar el DBMS, será posible indicar, en caso de que soporte tanto auto-incrementales, como secuencias, cuál de ellos se desea usar, mediante **useAutoIncrementId.**

Ejemplo:

```
@UaithneConfiguration(myBatisBackendConfigurations =
@MyBatisBackendConfiguration(useAutoIncrementId = Ternary.FALSE,
                    backend = MyBatisBackend.POSTGRE_SQL_9_0))
```

En una DBMS como la de POSTGRE_SQL_9_0 que soporta tanto auto-incrementales como secuencias, se puede obligar a utilizar secuencias mediante **useAutoIncrementId = Ternary.*FALSE***, y si se pusiera **Ternary.*TRUE*** se utilizarían auto-incrementales, aunque en caso de que se puedan usar auto-incrementales , ya los usará por defecto y por tanto no sería necesario especificar el valor de éste campo.

También puede configurarse el parámetro "**idSecuenceNameTemplate**" para especificar el nombre de la secuencia. Por defecto su valor es el String "[[table]]_[[column]]_seq", en este string se puede utilizar [[table]] o [[TABLE]] que representa el nombre mapeado (nombre en la base de datos) de la tabla y [[column]] o [[COLUMN]] que representa el nombre mapeado (nombre en la base de datos) de la columna (que representa el id) para la cual se genera el identificador.

Además se ha añadido la anotación **@IdSequenceName** que permite especificar el nombre de la secuencia sobre una entidad, afectando a todos los identificadores de la misma, o sobre un campo identificador para indicar el nombre de su secuencia. Tiene preferencia sobre el parámetro **idSecuenceNameTemplate**.

En caso de que se necesite especificar la manera en la cual se obtiene el valor actual y siguiente del ID, se debe utilizar la anotación **@IdQueries**, indicando como parámetros "**selectNextValue**", que se corresponderá con la query a ejecutar para obtener el siguiente valor del campo marcado como ID, y "**selectCurrentValue**", relativo a la query para la obtención del valor actual del campo. La anotación se puede declarar tanto en la entidad, como sobre el campo identificador y tendrá preferencia sobre la anotación **@IdSequenceName**.

En caso de que no se desee autogenerar el Id se ha de indicar en la anotación de la siguiente manera: @Id(autogenerated= false).

Configuración del bus

Uaithne facilita la creación de un bus que comunique las distintas capas de la aplicación mediante la conexión de componentes capacitados para alojar los distintos módulos de cada capa.

Una operación puede atravesar las capas de la aplicación que se deseen y el resultado recorrerá el camino inverso. El bus se crea mediante la combinación de las implementaciones de las interfaces **ExecutorGroup** y **Executor.**

Executor

ExecutorGroup

Un **Executor** representa una unidad que agrupa las operaciones (**módulo**), y tras recibir una, devuelve el resultado. Típicamente contiene una entidad y todas las operaciones asociadas con ella.
Un **ExecutorGroup** (o grupo) agrupa los módulos (Executors) de la aplicación, a modo de capas. Recibe una operación y se encarga de entregársela al ExecutorGroup de la siguiente capa o a la implementación de ejecución de operaciones del módulo que corresponda (Executor).

Con la intención de proporcionar una explicación concisa y sencilla, en la que los conceptos puedan quedar bien asentados en el usuario, se va proceder a llevar a cabo la configuración del bus mediante un ejemplo:

Para proceder a la definición del bus, se ha de crear una clase (p.ej. Bus.java) que contará al menos con los siguientes elementos:

- Un atributo privado que representará el punto de acceso (uno como mínimo) del bus de la aplicación, de tipo **ExecutorGroup**.

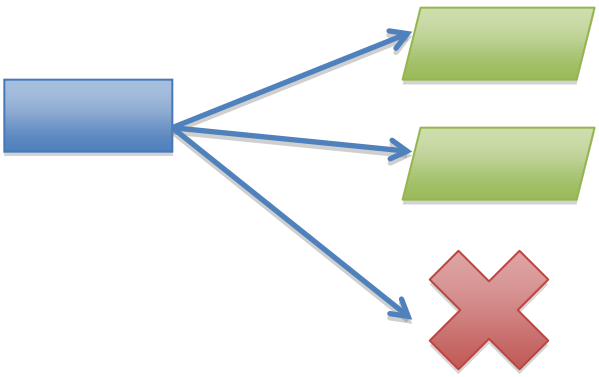
El punto de acceso de la aplicación (o también en el caso de que hubiera varios) se van a comportar siguiendo el patrón de diseño Singleton, por tanto serán necesarios al menos los siguientes métodos:

- Un método privado, estático y sincronizado (exclusión mutua mediante semáforos), que inicialice el/los bus/es. Ejemplo:

```
private static synchronized void init() {
   if (bus != null) {
      return;
   }

   ManagedSqlSessionProvider sessionProvider;

   try {
      sessionProvider = newManagedSqlSessionProvider("org/uaithne/samples/vuelo/DataBase.xml");
   } catch (IOException ex) {
      logger.log(Level.SEVERE, "Unableto load myBatisconfiguration", ex);
      return;
   }

   MappedExecutorGroup dbLayer =  new MappedExecutorGroup();
   dbLayer.addExecutor(new VueloMapper(sessionProvider));

   ManagedSqlSessionExecutorGroupmanagedSqlSessionExecutorGroup = new
ManagedSqlSessionExecutorGroup(dbLayer, sessionProvider);
   ValidateBeanExecutorGroupvalidateBeanExecutorGroup = new
ValidateBeanExecutorGroup(managedSqlSessionExecutorGroup);
   LoggedExecutorGrouploggedExecutorGroup = new LoggedExecutorGroup(validateBeanExecutorGroup,
logger, "uaithne vuelo");

   bus = loggedExecutorGroup;
}
```

Lo primero es comprobar que el bus está inicializado y si así fuera no volver a hacerlo.
Lo más común es que la última capa de una aplicación sea la de base de datos, por lo que es necesario obtener la sesión de base de datos. Utilizando MyBattis se debe crear una instancia de la clase **ManagedSqlSessionProvider** pasándole el archivo de configuración xml de la base de datos como parámetro.

Si es la última capa, se declara como un **MappedExecutorGroup**, que se encarga de delegar la ejecución al correspondiente **Executor** y en caso de que la operación no tuviera ninguno asociado se produce un error.



MappedExecutorGroup

Una vez definida la capa (dbLayer en el ejemplo), se le pueden asignar tantos **Executor** como se deseen, en el caso de MyBattis, tal y como se ha comentado, las clases anotadas con **@MyBatisMapper** generan una implementación del módulo (VueloMapper en el ejemplo) que se inicializa con la sesión de base de datos pertinente.
Uaithne también genera una capa que ofrece la posibilidad de ponerse por encima de la capa de base de datos como interceptor de las operaciones, con el fin de preparar la conexión a la base de datos y permitir realizar rollbacks (si la conexión lo permite) en caso de que alguna operación falle. Para ello se debe utilizar la clase **ManagedSqlSessionExecutorGroup** que extiende de un **ChainedExecutorGroup**, el cual permite delegar la ejecución de un **ExecutorGroup** a otro, modificando el comportamiento de la operación entre medias.

ChainedExecutorGroup

En el ejemplo también se puede observar una capa que lleva a cabo la BeanValidation de las operaciones, esta capa no es generada por Uaithne sino que se ha creado manualmente extendiendo la clase **ChainedExecutorGroup** y sobrescribiendo el método **execute** para interceptar la operación y llevar a cabo la validación de la operación de la siguiente forma:

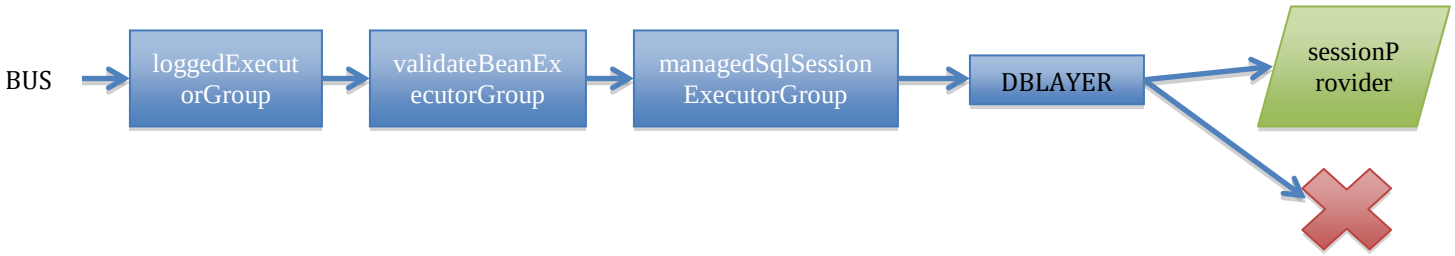
```
@Override
public<RESULT, OPERATION extends Operation<RESULT>> RESULT execute(OPERATION
operation) {
   Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
   Set<ConstraintViolation<OPERATION>> violations = validator.validate(operation);
   if (violations.size() > 0) {
      throw new IllegalArgumentException(violations.toString());
```

```
    }
    RESULT result = super.execute(operation);
    return result;
}
```

La capa más externa, en el ejemplo, se corresponde con el log de la aplicación, **LoggedExecutorGroup**y también es generada de forma automática.

Por último, una vez se han conectado todas las capas, se establece como punto de entrada la capa más externa (bus = loggedExecutorGroup;) de manera que al lanzarse una operación pueda seguir el camino correspondiente.

Una vez definido el bus tendría la siguiente apariencia:



- Un método de acceso al bus de la aplicación, que en caso de que no esté inicializado lo hace. Ejemplo:

```
public static ExecutorGroup getBus() {
    if (bus == null) {
        init();
    }
    return bus;
}
```

De esta manera cuando se desee ejecutar una operación se podrá acceder al bus de la aplicación mediante una invocación a getBus() y llamar al método **execute** del bus pasándole como parámetro dicha operación.

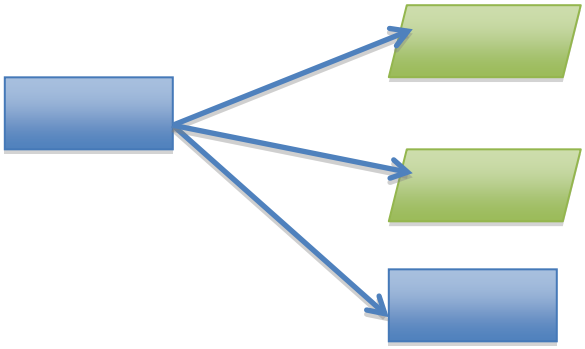
Ejemplo:

```
ExecutorGroup bus = Bus.getBus();
SelectPasajeroFromIdVuelooperation = new SelectPasajeroFromIdVuelo(new BigInteger("4"));
BigInteger pasajero = bus.execute(operation);
```

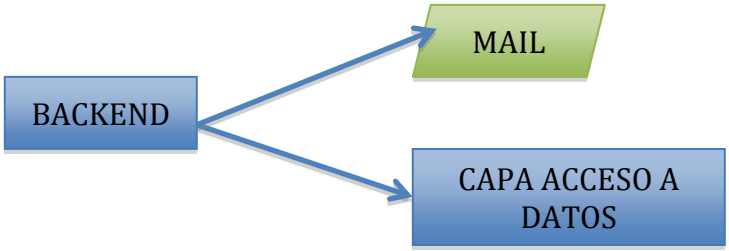
Otros elementos de configuración del bus.

ChainedMappedExecutorGroup

Consiste en un ExecutorGroup que delega la ejecución de una operación en el correspondiente **Executor**, pero en caso de que no tuviera ninguno asociado la delega en otro **ExecutorGroup.**



Por ejemplo, supongamos un módulo que se encarga del envío de e-mails dentro de la capa de backend y no requiere acceso a una capa inferior. Las operaciones destinadas el envío de e-mails llegarían a esa implementación del módulo y otro tipo de operaciones podrían ir a otra implementación si la hubiera o continuar directamente hacia capas inferiores.



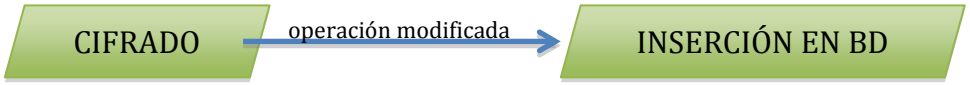
ChainedExecutor

Consiste en un módulo (**Executor**)que delega la ejecución de una operación en otro módulo. El primer módulo actúa como interceptor del segundo.

ChainedExecutor

Por ejemplo un caso de uso podría ser el caso en el que se desea cifrar la contraseña de un usuario antes de acceder a darle de alta en la base de datos. El primer módulo interceptaría la operación de inserción en base de datos tomaría el campo de la

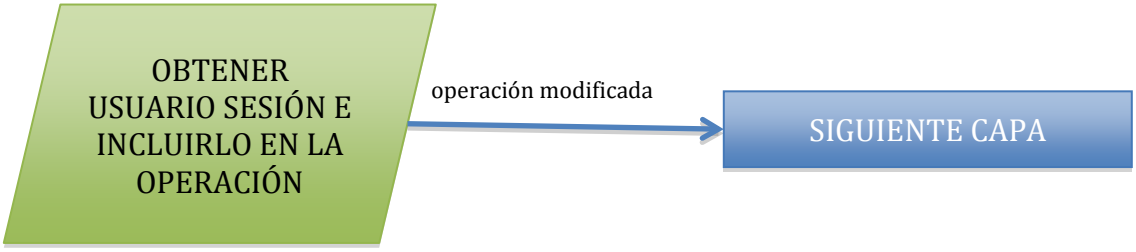
contraseña, pasaría a cifrarlo y modificarlo, y tras ello delegaría la ejecución al módulo que contenga la implementación de la inserción en base de datos.



ChainedGroupingExecutor
Un módulo (**Executor**) que delega la ejecución de una operación en un grupo (**ExecutorGroup**).

ChainedGroupingExecutor

Un posible uso podría ser a modo de interceptor de una operación para obtener, en la capa web, el usuario (actual) que la realiza, de la sesión y poder añadir el id con el que está autenticado.

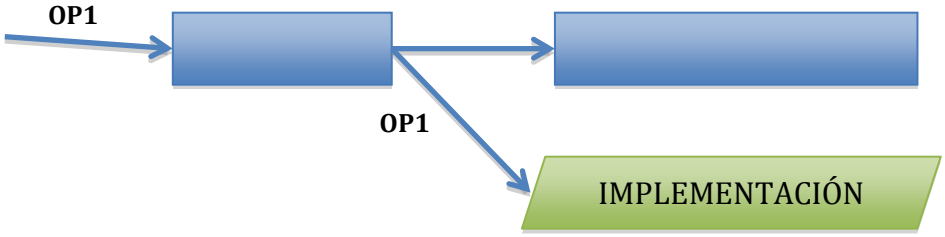


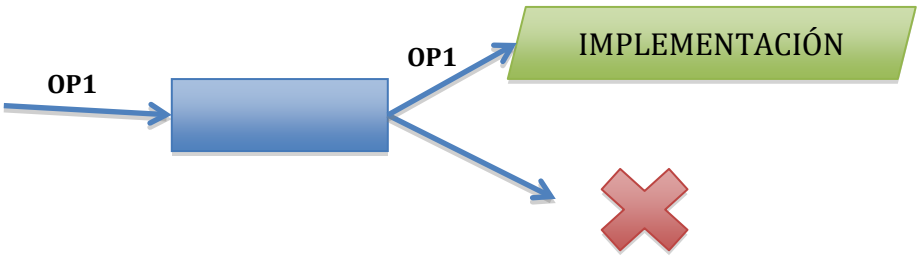
Configuración de comportamientos comunes

A continuación se muestran algunos de los casos típicos de configuraciones del bus en función del comportamiento que se desea obtener para una operación.

Implementa

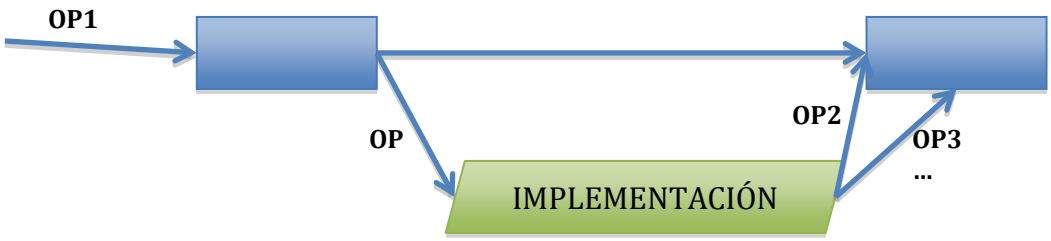
Es el caso más simple, en el que llega una operación sencilla, es decir, que no requiere llamar a otras operaciones, y se desea llevar a cabo su implementación.





Implementa y utiliza

En este caso la operación que se desea implementar requiere lanzar una o varias operaciones para poder llevar a cabo dicha implementación.



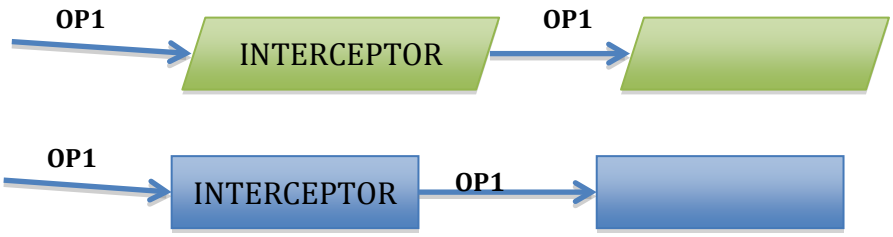
El resultado de las operaciones que lanza la implementación se utiliza para completarla.

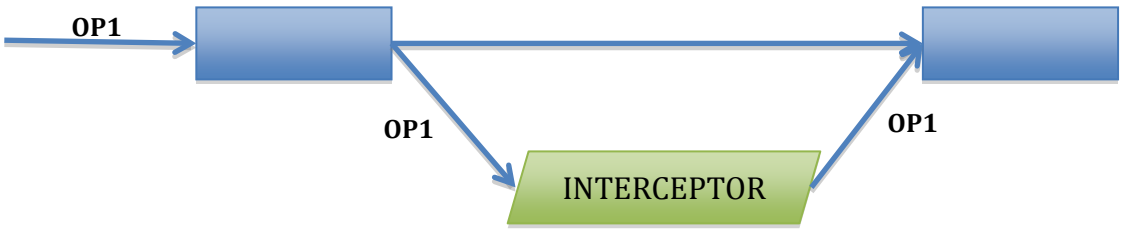
Intercepta

Un interceptor es un elemento que se interpone en el flujo de ejecución de la operación permitiendo modificar su comportamiento; por ejemplo, se puede decir que un trigger de base de datos actúa como un interceptor de las operaciones que se realizan sobre una tabla.

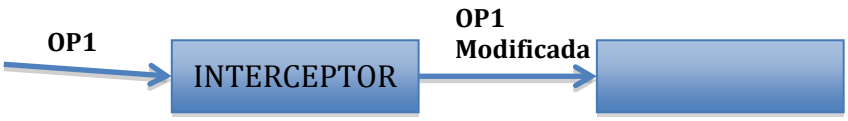
Las distintas funciones que puede realizar un interceptor son:

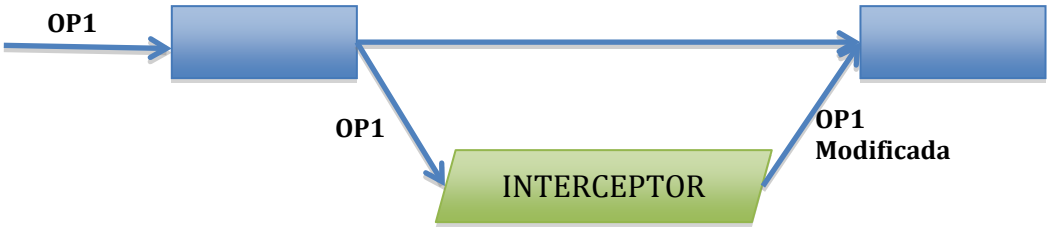
- Ser un mero **observador**, por ejemplo, un log. La operación pasa por él y no sufre cambios. (Se muestra un ejemplo con cada una de las posibles configuraciones de interceptor)



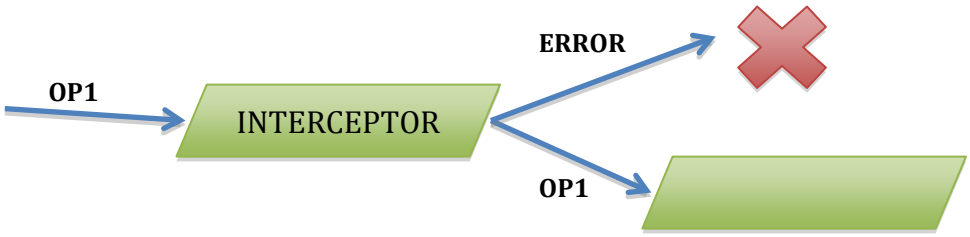


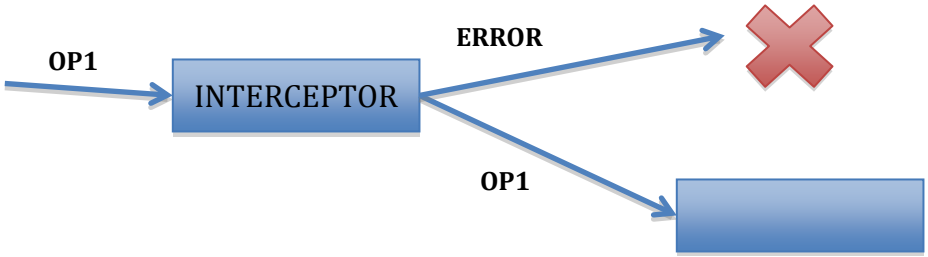
- **Modificador**. La operación se modifica y/o se completa. Por ejemplo, añadir la identidad del usuario a la operación, tomándola de la sesión.

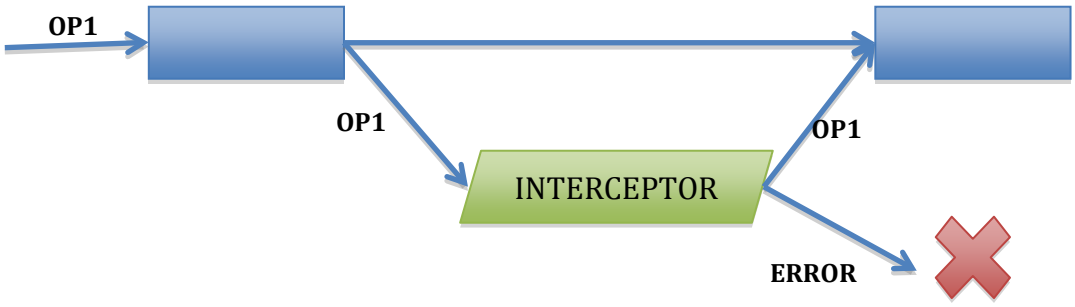


- 

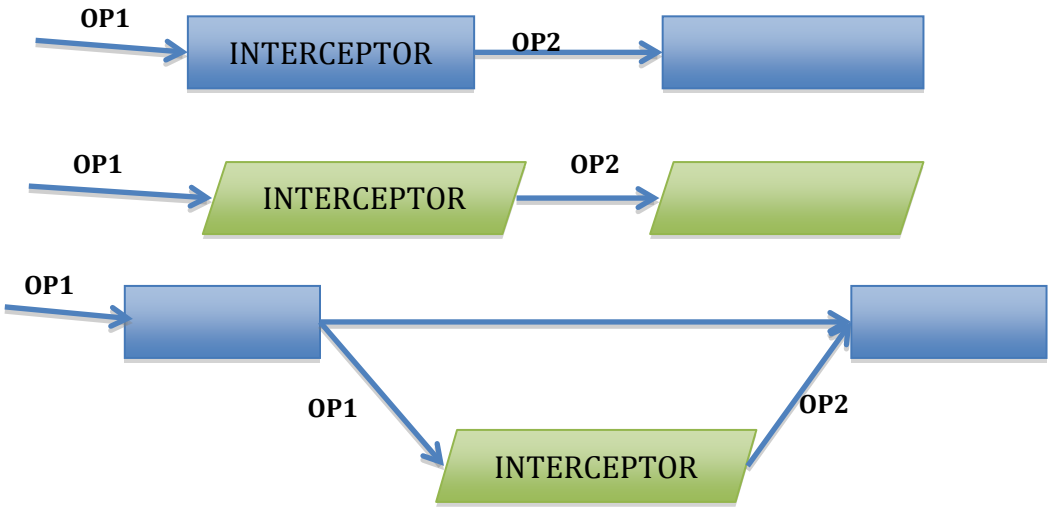
- **Controlador**. Decide si la operación continúa o no. Por ejemplo, se comprueba si se dispone de los permisos necesarios para ejecutar una operación.



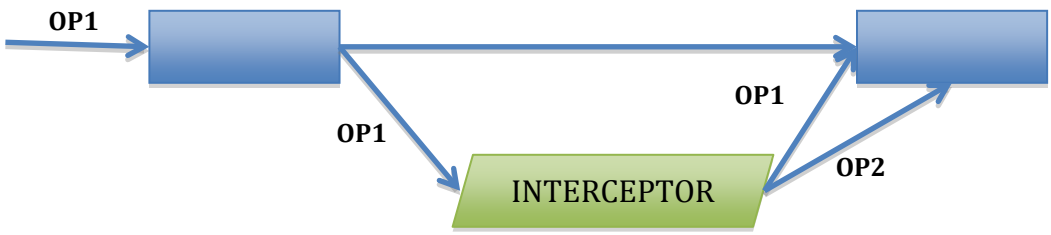




- **Suplantador.** Sustituye una operación por otra. La operación que sale no tiene por qué ser del mismo tipo que la que entra. Por ejemplo, si originalmente había una operación que se ha expuesto públicamente y no se desea romper la compatibilidad con ella, aunque el sistema interno a pasado por varias refactorizaciones que han dejado fuera de lugar a la operación; para mantener la compatibilidad se permite seguir ejecutando la operación pero ésta a su vez delega la ejecución a una segunda.



Estas funciones de interceptor que se han comentado son las más básicas; se pueden combinar entre ellas para crear interceptores más complejos.
Ejemplo de **controlador** combinado con **suplantador**:
Suponer el caso en el que los datos que retorna una operación provienen normalmente de una fuente de datos, pero en ciertas circunstancias los datos se deben extraer de otra fuente de información (por ejemplo, si no se encontró en la primera fuente de datos).


OP1
OP1
OP1
INTERCEPTOR
OP2